Extracted from:

# Mazes for Programmers

## Code Your Own Twisty Little Passages

This PDF file contains pages extracted from *Mazes for Programmers*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.



The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

# Mazes 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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

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

Maze-making seems magical when you're outside looking in, but don't be fooled. *There is no magic.* Starting on this very page, we'll begin demystifying the processes that drive maze generation. We'll see the scaffolding that lies just beneath their surface. We'll get specific, talking about what exactly mazes *are*, and then we'll get the ball rolling with two simple ways to create mazes, walking through them together with paper and pencil.

Eventually, this will take us to some exciting places, but like most beginnings, ours is quite humble. Here, it all starts with algorithms.

We're going to focus on those algorithms that produce mazes *randomly*. Passage length, the number of dead ends, crossroad frequency, and how often passages branch will all be determined by randomly choosing from a prescribed list of possibilities.

There is no universally ideal algorithm for generating mazes, so over the course of this book we'll explore twelve different ones. You'll learn how to choose between them depending on your project's needs, such as speed, memory efficiency, or simplicity (or even your own personal sense of aesthetics!). On top of that, most of the algorithms have little idiosyncrasies that cause the mazes they generate to share some feature, like short, stubby passages, or maybe the passages all skew a certain direction. We'll explore those, too.

But we'll get to that. By the end of this book you'll be an expert, able to nimbly switch between these different algorithms to choose just the right one for the job. You'll be pounding these out in code before you know it.

First, though, let's do it on paper.

**Joe asks:**

### What's an Algorithm?

An *algorithm* is just a description of a process. Like a recipe in a cookbook, it tells you what steps to take in order to accomplish some task. *Any* task. Algorithms exist for everything. If lasagna is your goal, then the steps you take to make lasagna are your algorithm. Want to make your bed, or drive to work? Both can be described as a series of steps. More algorithms! Algorithms launch rockets, land airplanes, drive cars, sort information, and search the Web. Algorithms solve mazes. And if you're out to *make* a maze, like we are, your algorithm consists of the steps you take to make that maze.

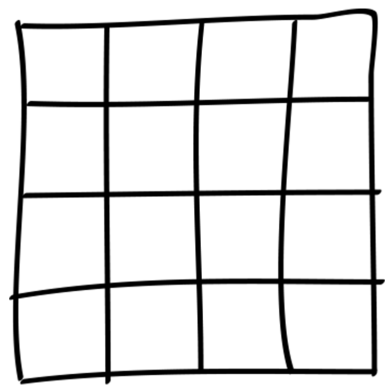
## Preparing the Grid

We're going to start by drawing a *grid*—just a regular crosshatching of perpendicular lines. This scaffolding will form the skeleton of the maze, the bones and sinews that will give structure and stability to our final product.

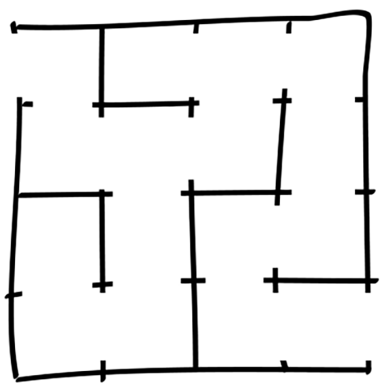
Here's what I want you to do.

Get out a piece of paper. It doesn't have to be fancy—a napkin will do in a pinch. You'll want something to write with, too, and erasability will be a plus.

On this piece of paper, draw a grid. Four-by-four ought to be plenty big enough for this first experiment, and don't worry about the lines being all neat. Anything like this figure should be fine.

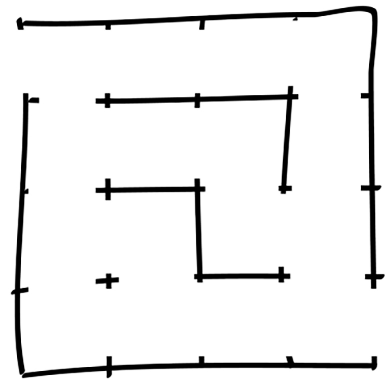


This is our starting point. We'll call the individual squares *cells*, and the grid lines around them *walls*. Beginning with this grid, our task is to erase just the right walls—*carve* just the right *passages*—in order to produce a maze.



That happens to be exactly what the algorithms in this book will do for us. Most of them create what are called *perfect* mazes, where every cell can reach every other cell by exactly one path. These mazes have no *loops*, or paths that intersect themselves. That's significant! This figure is an example of one of these perfect mazes.

Don't mistake the name for a value judgment, though. The "perfect" bit simply refers to its logical and mathematical purity. A maze may be perfect (mathematically), and yet flawed (for example, aesthetically), at the same time!



The opposite of a perfect maze is called a *braid* maze. These are characterized by few (if any) dead ends, and passages forming loops. Here's an example of a braid maze.

Going from one point to another in these kinds of mazes can be accomplished by multiple different paths, or solutions. We'll see more of them in Chapter 9, *Braiding and Weaving Your Mazes*, on page ?, but for now we'll focus just on their counterparts, the perfect mazes.

Let's create some!

### Labyrinths versus Mazes

Some people prefer "labyrinth." Others like "maze." Some even use the word labyrinth to refer to a particular kind of maze, a single passage that never branches but winds in a convoluted path from start to finish.

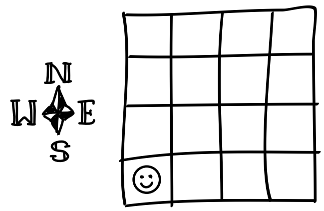
Ultimately, though, it doesn't matter what they're called. Labyrinth or maze, they (mostly) mean the same thing. I'll be giving preference to the word "maze" in this book. And while those non-branching versions (technically called *unicursal* mazes) are fun to play with, they are sadly beyond the scope of this book. We'll be focusing on *multicursal* mazes—those with branching passages—which will prove to be plenty all by themselves!

## The Binary Tree Algorithm

The Binary Tree algorithm is, quite possibly, the simplest algorithm around for generating a maze. As its name suggests, it merely requires you to choose between two possible options at each step. For each cell in the grid, you decide whether to carve a passage north or east. By the time you've done so for every cell, you have a maze!

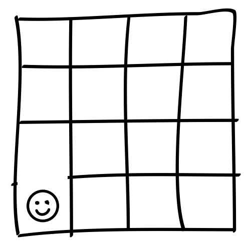
This process of looking at cells is called *visiting* them. Visiting them in some order is *walking the grid*. Some walks might be random, choosing directions arbitrarily from step to step, like the ones we'll see in Chapter 4, *Avoiding Bias with Random Walks*, on page ?. Others are more predictable. For Binary Tree, it turns out that we can do it either way. The algorithm really doesn't care what order we use to visit the cells.

Let's walk this together and see how the Binary Tree comes together in practice. I'll flip a coin at each step to decide which direction we ought to carve a passage. Also, while the Binary Tree algorithm itself doesn't care where in the grid we begin walking, for the sake of this example we'll just go with the cell in the southwest corner.

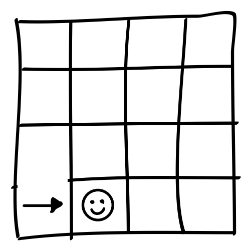



Our choice is this: do we erase that cell's northern wall, or its eastern wall? Let's see what the coin says. If it comes up heads, we'll carve north. Tails, we'll carve east.

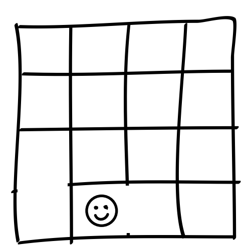
And…heads. Looks like we erase the northern wall.



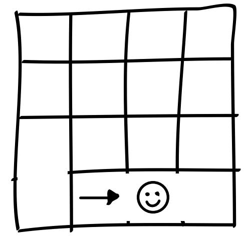
Note that although these two cells are now linked by a connecting passage, we haven't technically visited that second cell yet. We *could* choose to visit that cell next (because Binary Tree really doesn't care which order we visit the cells) but moving across a row and visiting its cells in sequence is simpler to implement. Let's wait and hit that northern cell when this row is finished. For now, let's just hop over to the one immediately to the east of us.



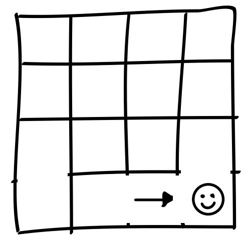
Flipping the coin here, we get tails. This means we'll erase the eastern wall of our current cell.



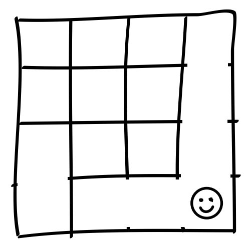
And flipping the coin for the next cell over gives us tails again.



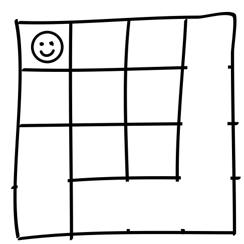
Moving east again, our current cell becomes the one in the southeast corner. We could certainly flip the coin here, too, but consider what would happen if the coin came up tails. *We'd have to carve a passage through the outer wall of the maze.* This is not generally a good idea. We'll talk more in a moment about adding entrances and exits to your mazes, but for now we want to avoid tunneling out of bounds. Since that effectively forbids going east, north becomes our only viable option. No need to flip a coin—let's just take care of business and carve north.



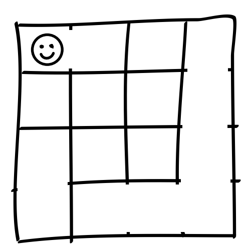
In fact, that constraint exists for every cell along that entire eastern boundary. None of them can host an east-facing passage. We might as well just take care of those now by carving north on each one of them. We'll consider each of them visited as well.



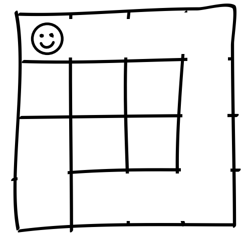
Now, for the sake of demonstration, let's jump all the way to the northwest corner and see what happens next. (Yeah, this is a bit unorthodox…but remember, Binary Tree only needs us to visit all the cells—it doesn't care what order we use to do that.)



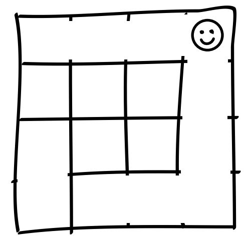
Once again, we could flip a coin, but consider what happens if the coin lands heads-up: we'd have to carve through that northern wall. We don't want that. Instead, we'll forego the coin flipping and just carve east.



Again, notice how that constraint applies to every cell along that entire northern boundary. You can't carve north from *any* of them, so *all* of them default to going east instead.

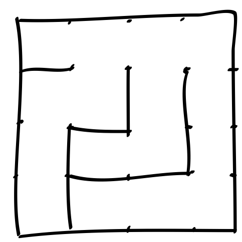


One more special case to consider. Let's jump to the northeast corner.



We can carve neither north, nor east from here. Our hands are tied. With nothing to choose from, we choose nothing. Of all the cells in our grid, this is the only one for whom nothing can be done. We shrug our shoulders and skip it.

Go ahead and grab your own coin, now, and flesh out the rest of those cells that haven't been visited yet. Once a decision has been made for every cell, you should be left with a maze that looks something like the figure.



That's really all there is to it! You just learned the Binary Tree algorithm for random maze generation. Painless!