The O-Maze-ing Caml

Alex Lin and Melissa Yu

April 27, 2016

The *O-Maze-ing Caml* is an OCaml-based application that randomly generates mazes and computes the solutions to them. The program also has graphical capabilities for rendering generated mazes onto the user's screen. In designing this project, we intentionally employed recursive algorithms to take advantage of OCaml's functional paradigm. The code can be found at https://github.com/al5250/the-o-maze-ing-caml.

1 High-Level Overview

We begin with a high-level description of our project before delving into the specific details within the code files. Section 2 address the Maze Generation portion of our program, while Section 3 focuses on Maze Solving.

1.1 Code Structure

The code is divided into three files:

- `main.ml` - interprets user input and executes the relevant functions of the program
- `cell.ml` - contains the Cell module, which provides implementation for the individual square units that the maze is composed of
- `maze.ml` - contains the Maze functor, which provides implementation for maze generation and maze solving based on the Cell module; this file contains the bulk of the project code

This division was enforced to logically separate the creation of maze modules (`cell.ml`, `maze.ml`) from their use (`main.ml`). Furthermore, we noted that mazes of different shapes and sizes could be created based on the properties of their cells. Thus, we packaged the implementation for cells separately from the implementation for mazes; this led us to naturally create a Maze functor that inputed modules of type CELL and outputted modules of type MAZE.

In Maze, we also render the maze and its solution onto a graphical interface as a side effect of the `generate` and `solve` functions. Initially, we considered factoring all of the drawing functions into a separate module, yet we later decided that there was no need for this extra layer of abstraction, as users do not have much of a reason for creating mazes but not viewing them.

1.2 Running the Program

To run the program, enter `./main.byte` into the command line. You will be prompted with

```
# new maze?  (y/n):
```

Enter 'y' for yes and you will see

```
# enter a difficulty (between 1 and 7) to generate maze:
```

Entering the number k will prompt the program to randomly generate a maze of size 2^k by 2^k cells.

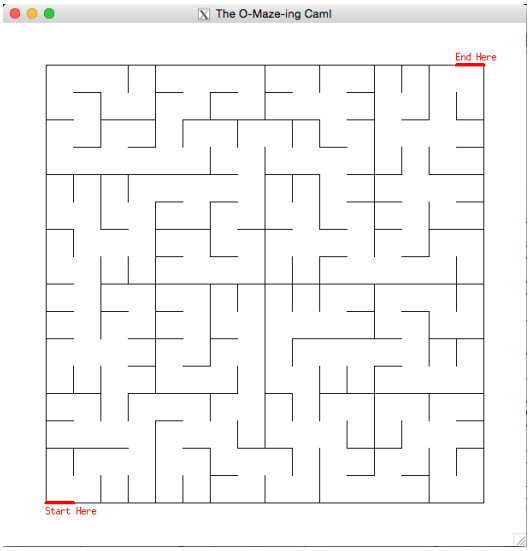

Figure 1: An example of a randomly generated 2^4 x 2^4 maze.

After the maze renders, you will see

```
# see solution?  (y/n):
```

Enter 'y' for yes and you will see the solution to your randomly generated maze drawn in red.

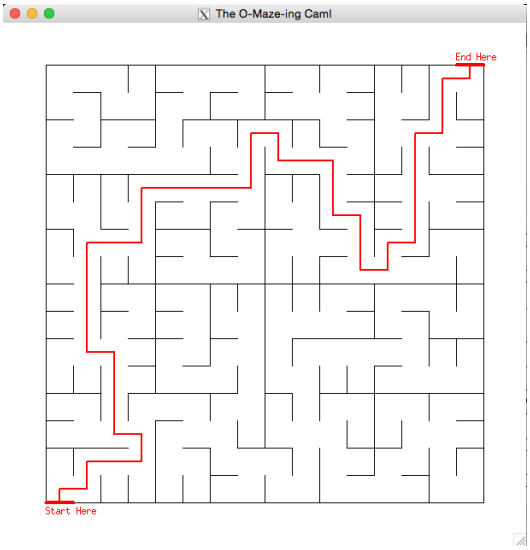

Figure 2: The solution to the above maze.

The program will then loop back to the beginning with

```
# new maze?  (y/n):
```

Enter '`y`' for a new maze and '`n`' to quit the program.

2 Maze Generation

The first of the two major components of our program addresses how to randomly create perfect mazes, given the dimension of the maze. By *perfect maze*, we mean a maze that only has one possible solution.

2.1 Recursive-Division Algorithm

The recursive division algorithm is the fundamental idea behind how our `generate` function in `maze.ml` works. Although there are many algorithms for maze generation, such as Prim's, Kruskal's, Depth-First Search, etc., we chose recursive division, because it works well with OCaml's functional paradigm. Mazes are comprised of cells. Each cell has four sides - top, bottom, left, and right. Adjacent cells share a side. Each side can either be *closed* (i.e. there is a wall between the two adjacent cells) or *open* (i.e. there is no wall and one can move freely between the adjacent cells).

Let's say we want to generate an n by n maze. The algorithm begins with a single square cell of size n by n. This single cell is then divided into four $\frac{n}{2}$ by $\frac{n}{2}$ cells. To preserve the maze structure, only one of the four inner cell sides are kept as a wall; the other three are opened.

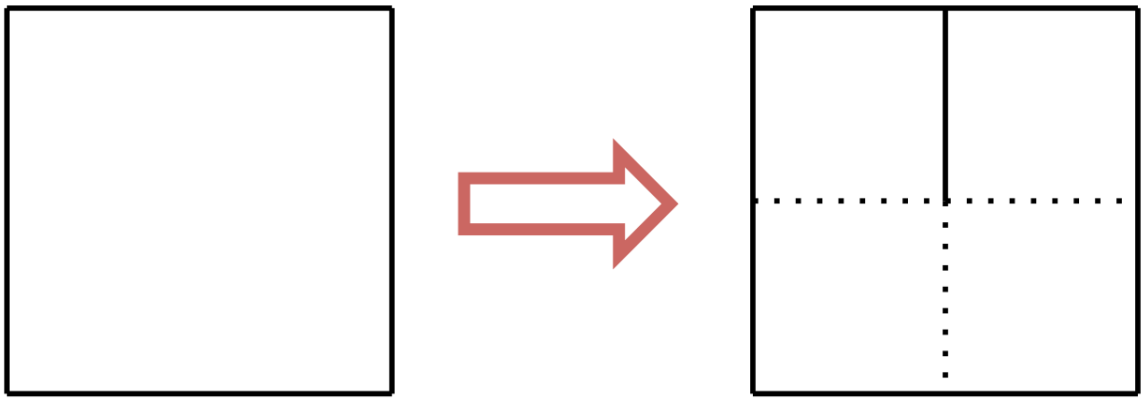

Figure 3: A cell is divided into four smaller cells. Only one inner side is kept closed.

Then, the algorithm recursively goes through the same process on each $\frac{n}{2}$ by $\frac{n}{2}$ cell. During the recursive division process, if an open side of length m is split into two sides of length $\frac{m}{2}$, only one of these sides can be open; the other must be closed. We do this to preserve the perfect maze invariant; at any point in the algorithm, there can only be one way to get from any cell to any other cell.

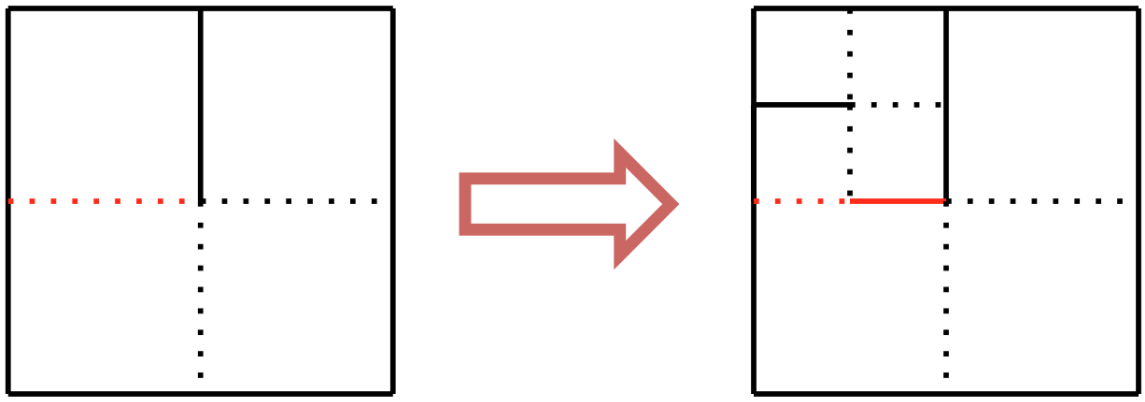

Figure 4: An open side is divided into 2 smaller sides - one closed and one open (see red).

We hit the base case when the length of the cell side is equal to 1. At this point, we stop dividing and return the resulting perfect maze of n by n cells. We make two sides on the outer boundary open to have a start point and an end point.

One advantage of this algorithm is that it is optimally efficient; to produce a maze of n^2 cells, it takes $O(n^2)$ time. Let $T(n)$ be the time it takes to create a n by n maze using recursive division. The recurrence relation is then

$$T(n) = 4T\left(\frac{n}{2}\right) + c$$

where c is a constant representing the constant work done at each division. Unraveling the recurrence, we have

$$\begin{aligned}
T(n) &= 4T\left(\frac{n}{2}\right) + c \\
&= 4\left(4T\left(\frac{n}{4}\right) + c\right) + c \\
&= 16T\left(\frac{n}{4}\right) + 5c \\
&= 16\left(4T\left(\frac{n}{8}\right) + c\right) + 5c \\
&= 64T\left(\frac{n}{8}\right) + 21c \\
&\vdots \\
&= 4^k T\left(\frac{n}{2^k}\right) + (1 + 4 + \ldots + 4^{k-1})c
\end{aligned}$$

We stop when $2^k = n$, or $k = \log_2 n$. Thus,

$$T(n) = n^2 + (1 + 4 + \ldots + 4^{\log_2 n - 1}) = n^2 + \frac{4^{\log_2 n} - 1}{3} = n^2 + \frac{n^2 - 1}{3} = O(n^2)$$

Hence, we have shown that Recursive Division is optimally fast, because it takes $O(n^2)$ time to produce n^2 cell. Thus, when we made the design decision to implement Recursive Division over other algorithms such as Prim's, Kruskal's, etc., we did not sacrifice efficiency and simultaneously enabled our program to take advantage of OCaml's functional structure.