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.

Contents

1 Overview **2**
- 1.1 Code Structure . . . 2
- 1.2 Running the Program . . . 2

2 Maze Generation **3**
- 2.1 Recursive-Division Algorithm . . . 3
- 2.2 Design & Implementation . . . 5
- 2.3 Functions Explained . . . 6
- 2.4 Rendering Graphics . . . 7

3 Maze Solving **7**
- 3.1 Recursive-Backtracking Algorithm . . . 7
- 3.2 Design & Implementation . . . 8
- 3.3 Functions Explained . . . 9
- 3.4 Rendering Graphics . . . 10

1 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 `make` followed by `./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. 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. 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.

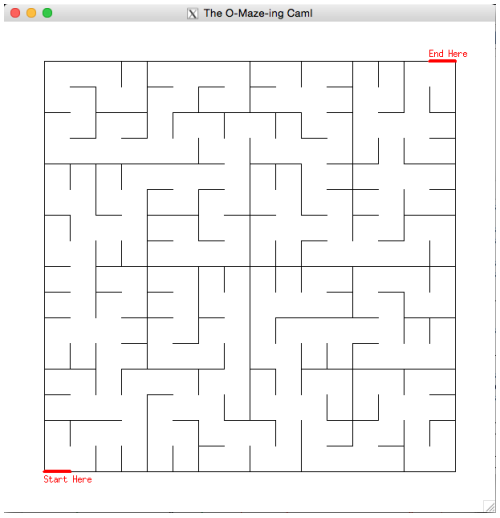

Figure 1: An example of a randomly generated 2^4 x 2^4 maze.

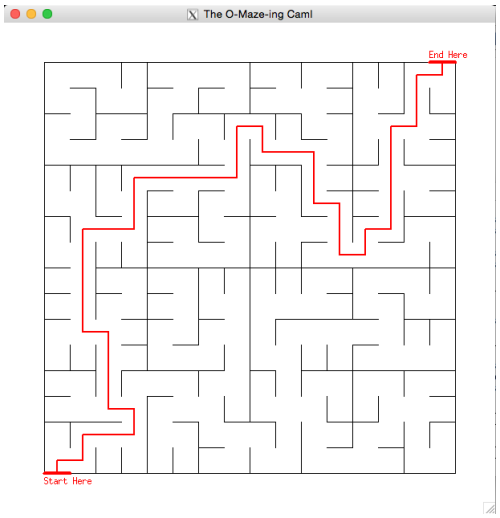

Figure 2: The solution to the above maze.

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.

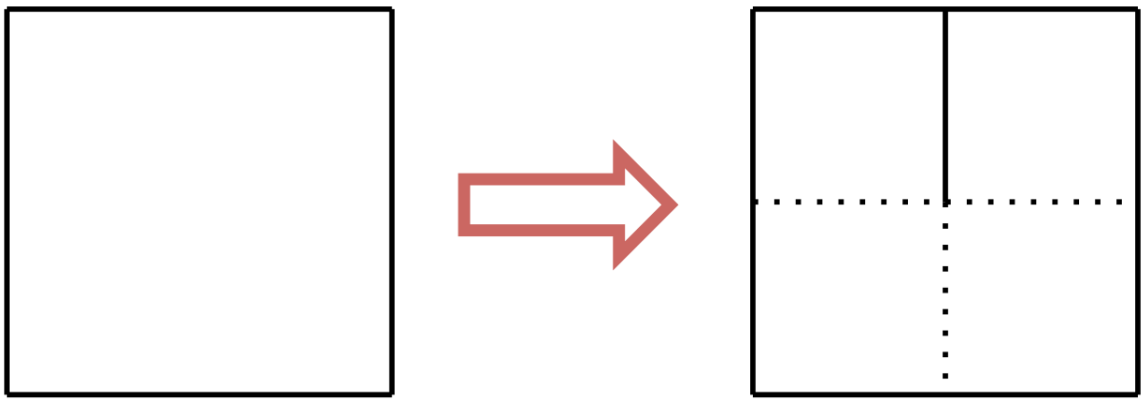

Figure 3: A cell is divided into four smaller cells. Only one inner side is kept closed.

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.

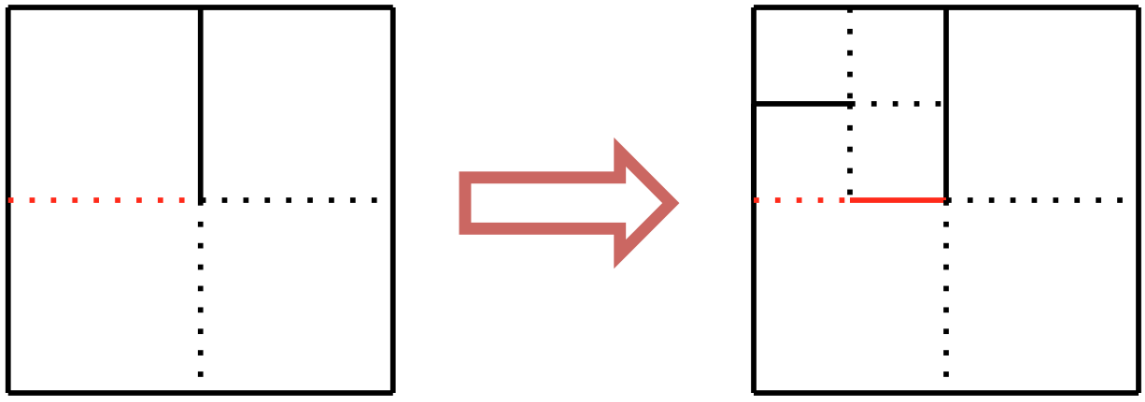

Figure 4: An open side is divided into 2 smaller sides - one closed and one open (see red).

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 cells. 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 any efficiency and simultaneously enabled our program to take advantage of OCaml's functional structure.

2.2 Design & Implementation

A natural implementation of the recursive division algorithm is to have `type cell` be a record that packages relevant variables such as `pos` (the double indexed position of the lower-left hand corner of the cell in the maze), `length` (the length of one side of the cell), and booleans for each side of the cell to indicate open (`true`) or closed (`false`). However, if we look at a single side of a cell, we see that it is also the side of another, adjacent cell. Thus, from the perspective of space efficiency, there is no reason to include this information twice - at least for the process of maze generation. Therefore, we made the design decision of letting each cell c only keep track of its left and bottom sides (see `cell.ml`). The cell above c would keep track of c's top side as its own bottom side and the cell to the right of c would keep track of c's right side as its own left side. In this fashion, the only sides left untracked are the ones on the outermost boundary of the maze, but these can just all be closed until the algorithm terminates. Then, two of them will become open for the start and end points of the maze.

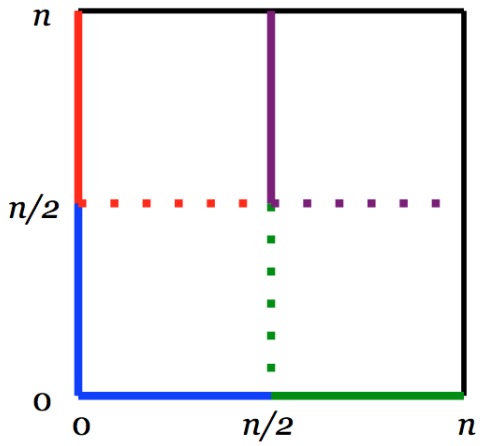

Figure 5: The abstract representation of mazes in our program. Each color is a different cell. For example, the green cell is {`pos = (n/2, 0); length = n/2; left = true; bottom = false`}.

A more subtle benefit of having each side tracked by only one cell becomes apparent during the process of recursive division. Let's say we have an open side s that is the right side of cell c and the left side of cell d. During the next step of recursive division, s needs to be split into two smaller sides - s_1 and s_2. As explained earlier, exactly one of s_1, s_2 must be open and the other must be closed, and this will be decided randomly. The cell-dividing algorithm will be recursively called on c and d separately, but we need to make sure that in each call, the same smaller side (either s_1 or s_2) is chosen as open and the same smaller side is chosen as closed. If both c and d kept track of s, then enforcing such an invariant would be computationally difficult and inefficient. On the other hand, by letting only d keep track of s as its left side, we avoid this issue entirely, because c will not process s at all. This was another factor that played into our decision to let each cell only keep track of its left and bottom sides.

With the structure of a cell set in place, we can then simply let `type maze` be a `cell list`. Initially, the list will just contain a single cell with side length n. In each iteration of recursive division, the algorithm will process a list of cells with side length k and return a list of cells of side length $\frac{k}{2}$.

2.3 Functions Explained

Maze generation utilizes functions from both modules of type CELL and modules of type MAZE. The main functions which generate a random maze are found in `maze.ml`:

- `initialize` - Creates an empty maze of size n, i.e. a single cell at (0, 0) with side n, by calling the cell module `C`'s generate function.
- `generate` - Main function used for maze generation. Creates a n by n maze of cells by recursively calling the cell module `C`'s `divide` function on the initial cell returned by `initialize`. Returns the resulting maze as a `cell list` and draws the maze as a side effect.

The functions in `cell.ml` are recursively called to do the work of actually splitting the cells:

- `generate` - Creates a cell with the specified position and dimensions, with the bottom and left walls closed by default.

- `divide` - Divides a single cell into four smaller cells of equal size in two steps. Let's call the original cell c. `divide` does the following:

  1. Bisects the bottom and left walls of c to create c_i. If the wall being bisected was originally open, `divide` preserves the perfect maze invariant by calling `modify`, which chooses one of the two halves of the original wall to remain open (for more detail, see the discussion in 2.1).
  2. Randomly sets three of the four inner walls of the divided cell to be open, leaving the remaining one closed.

- `modify` - Randomly chooses one of two input cells to apply the specified function `f` to.

2.4 Rendering Graphics

Once generated, drawing the maze is straightforward. Using the OCaml `Graphics` module, which includes functions for creating a new graph, drawing a line between points, and rendering text, we simply initialize an empty graphics window and then iterate through all cells in the maze, drawing each one. The relevant functions are found in `maze.ml`:

- `display_screen` - Initializes an empty graphics window with a square maze frame and title

- `draw_cell` - Draws a single cell by checking the left and bottom walls of the cell and drawing a line between the current position, (x, y), and $(x, y+dy)$ if the left wall is open, or $(x+dx, y)$ if the bottom wall is open. dx and dy are the scaled lengths of a cell on the display screen. Note that all positions encoded in a cell assume a cell has side length 1 pixel. When rendering a maze, these positions are scaled by a factor determined by the window and maze size.

- `draw_maze` - Iterates through all cells in a maze and calls `draw_cell` on each one. Although each iteration of `draw_cell` only draws the left and bottom walls of any given cell, iterating through all cells renders the complete picture, since the top and right walls of any given cell are the bottom and left walls of the adjacent cells, and we have already drawn the outermost four walls of the maze in `display_screen`.

- `draw` - Main function for drawing unsolved mazes; calls `display_screen` and `draw_maze`

3 Maze Solving

The second part of our program focuses on the solving of mazes (i.e. how to find a set of cells that give us a legal path from the starting point of the maze $(0, 0)$ to the ending point (n, n), where n is the dimension of the maze). Again, we chose to implement a recursive algorithm that is "OCaml-friendly".

3.1 Recursive-Backtracking Algorithm

The recursive backtracking algorithm is essentially a variant of Depth-First Search (DFS) applied to mazes. Conceptually, it works by going deeper and deeper along a random path in the maze until it hits a dead end. At this point, the algorithm backtracks along the current path of exploration until it finds another path to explore. The algorithm terminates when we reach the final cell, situated at the position $(n-1, n-1)$.

Envisioning the maze as a graph helps us see why recursive backtracking - though simple and often implemented by maze-solving 3-year-olds - is very efficient. Let each cell in the maze be represented by a vertex. Two vertices are joined by an edge if and only if they are adjacent and there is an open side between them. The perfect maze invariant necessitates the existence of one and only one path from any cell to any other cell in the maze. Hence, the graph representation of the maze must be acyclic. Furthermore, the graph must be connected, because every cell is accessible from every other cell. This means that any perfect maze, being connected and acyclic, is a tree.

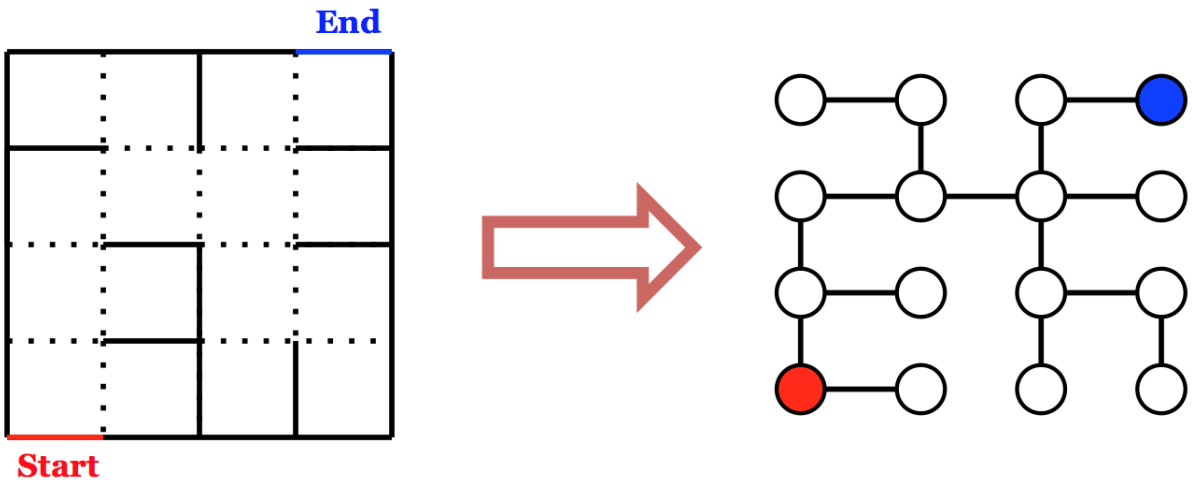

Figure 6: An example of the tree representation of a perfect maze. The goal is to find a path from the red vertex to the blue vertex.

Thus, running recursive backtracking on a maze is equivalent to running DFS on a tree. DFS has an asymptotic running time of $O(|V|)$ for trees - where $|V|$ is the number of vertices - and is optimally efficient for finding a path between two vertices. This means that recursive backtracking is also optimally efficient for maze solving (at least in an asymptotic sense). To find the solution to a maze of n^2 cells, the algorithm will take $O(n^2)$-time. Thus, our choice to implement this algorithm is supported by the fact that 1) it is efficient and 2) it is recursive and works well with OCaml.

3.2 Design & Implementation

In order to implement recursive backtracking/DFS efficiently, we needed to make sure that each cell had constant time access to each of its neighbors. Note that this is not possible for `type maze`, because each `type cell` only keeps track of itself and the state of the its left and bottom sides. Thus, we needed a new data structure to represent the maze. We chose to use an array, because mazes themselves are already set up like arrays. Furthermore, to move from a cell (i, j) to an adjacent cell, we can simply increment or decrement i or j by 1, which can be done in constant time.

Our new representation of a maze cell is called `type array_cell` and just keeps track of four booleans to indicate which of the four sides are open/closed. Then, a `type array_maze` is just a

two-dimensional array of `type array_cell`'s. The position of each cell in the maze is simply the double index of the corresponding `array_cell`.

Let `m` be an `array_maze` with dimension n. We implement recursive backtracking by starting from `m.0.0` and ending with a path to `m.(n-1).(n-1)`. We use two stacks (known in OCaml as `lists`) to accomplish this goal. The first stack is called `frontier`. It is a set of unexplored cells that are adjacent to/accessible from currently explored cells. The second stack is called `path`. It is a set of adjacent cells that incrementally build the solution to the maze from $(0, 0)$ to $(n-1, n-1)$. When the program terminates, `path` will be returned. Initially, only $(0, 0)$ is in `frontier` and `path` is empty. During the path exploration phase of the algorithm, cells will be repeatedly popped off of `frontier` and placed in `path`. If we hit a dead end and need to backtrack, cells will be popped off of `path` until we arrive at a cell that is adjacent to the next cell in `frontier`, so that we can start a new path of exploration. Section 3.3 gives further details on these functions. We chose to use this two-stack method, because it allows us to process each cell in the DFS algorithm in constant time.

3.3 Functions Explained

To solve the maze, we take advantage of the OCaml `Array` module, which provides methods for creating n-d matrices and accessing and modifying their elements in place. Two new types, `array_cell` and `array_maze` are defined in this section to simplify the solution process. Because all cells are the same size after recursive division has concluded, the length parameter becomes redundant, and storing the resulting cells in a matrix allows us to drop the position field as well. Let `array_cell` be a four-field record encoding boolean values representing whether each of the four sides of a cell is a wall. Then, the maze is represented by an `array_cell array array`. All relevant functions for solving mazes are found in `maze.ml`:

- `to_matrix` - Converts a `cell list` to an `array_cell array array` representation. Let the current cell being inserted into the array be c, with `c.pos` $= (x, y)$. Then the corresponding array_cell in the array representation is found at the index (x, y). For each cell popped from `maze`, we update the left and bottom side values of the corresponding cell in the matrix, as well as the right and top side values of the 2 adjacent neighbors.
- `is_neighbor` - Checks whether 2 given cells communicate (i.e., are both adjacent and directly accessible to each other).
- `get_neighbors` - Returns a list of all communicating cells to the given cell by calling `is_neighbor` on the cell's adjacent cells.
- `explore` - Extends the path forward one cell in each iteration, until either hitting a dead end, in which case `backtrack` is called, or reaching the exit cell in the maze, in which case we exit with the final solution path. Let the frontier be the list of unexplored communicating cells adjacent to our current solution path. In each iteration, we pop 1 element off the frontier and check it against the exit conditions. If the conditions are not met, we add the current cell's communicating neighbors to the frontier, rinse, and repeat.
- `backtrack` - Backtracks one cell along the current solution path until a new path of exploration is found. `backtrack` pops cells off the solution path until reaching a cell that communicates with the next element in frontier, gauged by calling `is_neighbor`. At this point, the latest new branch of exploration has been reached, and the function exits back to `explore`.
- `find_path` - Finds a solution path for a given `array_maze` by initializing `frontier` and `path` and calling `explore`.

- `solve` - Main function used for maze solving. Calls `to_matrix`, `find_path`, and `draw_solution` on the given maze, and returns a list of the cells in the solution path in reverse order (i.e., starting with the last cell).

3.4 Rendering Graphics

Because drawing the maze is a side-effect of generating one (and thus solve maze will never be called without first having drawing maze), rendering the solution simply requires us to draw the solution path on top of the already rendered graphics. Once again, we use the OCaml `Graphics` module. All of the functionality is contained in the function `draw_solution`, which connects the centers of successive cells in the solution path with line segments.