Mazes and Stacks

Andrew Rosen

Abstract

This lab will teach you about Mazes and Stacks and recursion and playing with Graphics, all the while knowing nothing about graphics or recursion. It also will test your ability to work with unfamiliar code-bases and platforms.

1 Brief Problem Summary

I've provided you with a few files to generate a maze. Your assignment is to create a basic maze solving algorithm to find it's way from the top left corner of the maze to the bottom right. If we're being generous we might say we're making a very basic AI.

To do so, you need to finish implementing `solveMaze()` method to perform a *depth-first search* using a stack.[1] Don't freak out if you don't know anything about graphics. The most complicated thing you will be doing is changing the colors of a few squares.

2 Basic Maze-Solving Algorithm

As I described in class, depth-first search is fairly straight-forward. When you have a choice in what direction to travel, make a choice. Follow that corridor and choose a new branch to go down as needed. If you reach a dead end, either because you're blocked by walls or there's nowhere new to visit, you backup until you find a new route to explore.

The way we do this with an algorithm is a stack.

```
push start position on top of stack
while maze exploartion is not done and and stack isn't empty
    peek to get our current position
    if we can go north and haven't visited there yet
        push the location to the north on the stack
        mark the current location as visited
    else if we can go south...
    repeat for east and west
```

[1] A term that might be worth looking up on wikipedia.

```
    else
        we can't go anywhere so we are at a dead end
        mark current as a dead end
        pop off the stack
```

Lookup depth first search for more details. The way this works for our code is that locations are represented by `Cell` objects and we can mark them visited by coloring them.

3 A Tour Of The Source Code

Please watch the companion video and the lecture for more details.

3.1 Maze

Run this class since it has the `main` method in it. It creates the `MazeGridPanel` and passes in parameters to set up how large of a maze you want. I've found that anything above a 100 × 100 maze is pretty slow.

3.2 MazeGridPanel

This holds our actuals maze. Our maze is held in the 2D array of `Cell` objects, called maze.

`solveMaze()` is your assignment and `genDFSMaze()` is your extra credit. `solveMazeQueue()` solves the maze using breadth-first search. `visited()` will check if the `Cell` at `row` or `col` has been visited by looking at the color (you should read this method.). `genNWMaze()` is the method I wrote that actually creates the maze.

3.3 Cell

The maze is made up of individual pieces of a grid, each represented by a `Cell`. Each cell has a `boolean` for each of the four possible walls it can have in any direction, as well as a row and col for easy reference to its location in the maze.

We'll keep track of whether we've seen a `Cell` by coloring it. My code considers white and red cells unvisited (red is used to mark the exit of the maze). A cell colored anything else has been seen or visited in some way, shape, or form. We can change the color of a `Cell` using the `setBackground` method, which takes in a `Color`. We can retrieve the color of a `Cell` using the `getBackground()`.

4 Extra Credit: Maze Generation

Complete `genDFSMaze()`, which will build a maze by using depth-first search. You can find the algorithm by clicking on this sentence, which is also a hyperlink.

5 Grading

A partially working solution is worth 50 points. A fully working solution is 100 points. The extra credit problem is worth 5 points.