Maze Generation In Haskell and Java

Anthony DuPar - adupar@ucsc.edu
Jordan Hyman - jahyman@ucsc.edu

March 21, 2014

Motivation:

The previous quarter, we were both enrolled in CS146, a Game AI class. A portion of this class focussed on pathfinding, and more specifically maze solving. When we were assigned to do a project this quarter, we initially were unsure what to do. After noticing a student from a previous quarter had done maze pathfinding, we immediately recalled our previous work with mazes and so decided to focus on them. Initially, we had planned to write a maze generating program in Haskell, with a program in Java which would then solve said maze. Due to unforeseen difficulties and overconfidence in our ability with Haskell, we decided to change the scope of our project from maze generation and solving to purely maze generation. Now we have two programs, one in Java and one in Haskell, which both generate mazes that we shall compare. We chose Haskell because we had been working with it all quarter and it was like no language we had ever experienced programming with before, so to be able to compare it with a language we're far more experienced with (Java) would prove to be an educational experience. Unlike Haskell, we both have had far more experience with Java, due to most of our previous classes mainly focussing on them.

The Maze Generation Algorithm and Our Modifications to it:

The maze generation algorithm we used was a depth first search algorithm that goes as follows:

i) Create three 2D arrays, one for eastern walls, one for southern walls, and a third for all visited cells.

ii) Starting from the upper left corner of the maze, we make the cell for that point in the visited array True, symbolizing that that point has now been visited.
iii) Obtain a list of cells neighboring the current cell we are in.
iv) Randomly select one of the neighbors in the list and check if it has been previously visited (it's corresponding location in the visited aray should be True).
v) If the neighbor has not been visited, the wall between these two cells is deleted. i.e. If the eastern wall between (0,0) and (1,0) is torn down, then east[0][0] is changed from True to False.
vi) The neighbor is then se as the current cell
vii) Repeat his process until all cells have been visited.

With this algorithm, we initially ran into a common error: on occasion, our current cell would be surrounded by neighbors which were all visited, causing our program to loop infinitely. To fix this, we created variables which stored the previous (x,y) coordinates, so that should this error occur, the current cell could backtrack to a cell where (hopefully) all of it's neighbors weren't already visited. A problem with this fix though, was that if the previous cell also had it's neighbors visited, then we would go back to the cell we were at prior (the first cell with all visited neighbors). We considered creating a list of previously visited cells, but couldn't seem to implement it correctly. Instead, when surrounded by already-visited neighbors, the current cell would be changed to a random cell which had been previously visited. This way, should he neighbors be all visited, we can move (eventually) to a cell where such isn't the case. Due to this, the maze isn't generated purely through DFS, but it's close enough and always generates solvable mazes. Once the maze is generated, different sets of characters are printed depending on the corresponding boolean value within the southern and eastern wall arrays. (i.e. south[0][0] == False && east[0][0] == False results in (0,0) having no eastern or southern wall, hence it'd be printed " ". Were both it's east and south wall booleans set to True, it's be printed as "_|").

The Haskell Code

Considering our initial project scope had been to have the Haskell program generate mazes for a solver written in Java, we worked on this portion first. Initially our maze was structured using Points and Cells, with Points being a Tuple of two Ints and Cells essentially being a Quadratic Search Tree which contained cells (i.e. Hall (0,0) Wall Wall Wall Wall symbolizes Cell (0,0) having a wall in each direction). We attempted to generate the maze using a DFS algorithm similar to our final one, though the idea to have 2D arrays for our southern and eastern walls had not occurred to us at the time, and instead we attempted to have our maze outputted as a single Cell(Point) which would then be explored much like a search tree. Though a problem with this meant that the northern "leaf" of a cell would have that cell as a southern leaf, which would then display it's northern leaf as that cell, so on and so forth, ad infinitum. Put simply, this wouldn't end up working very well.

After this initial defeat with Haskell, we decided to change the scope of our program. After all, if our Haskell code couldn't generate mazes, then what would our Java program be able to solve? After creating a program which generated "sort of mazes" on occasion in Java, we eventually wrote a superior program in Java which successfully created a solvable maze every time (with the rare exception of the occasional infinite-loop. No idea what causes it). From there, we gained a better insight on to how to write the Haskell equivalent. Using our new algorithm (stated in the previous section), we generated lists of lists for our visited cells, and eastern/southern walls. Unlike Java though, we were unable to change these lists due to their immutability. To go around this, we created a function which returned a new list that contained whatever new value we wanted changed along with whatever values the old list previously had. We also ran into problems when it came to using if-statements like we had in Java. Usually this was simply due to Haskell getting parsing errors, though we found it easier to simply use pattern-matching instead. Though considering we needed to keep track of so many variables, these lines

often got rather long. Funnily enough though, these functions were often immensely short in terms of what they needed to do, with the only exception to this of course being our generation. Since Haskell has no for or while loops, our generation had to be done through recursion. Via pattern-matching, we determined what direction the randomly selected neighbor came from. If this chosen cell had not been visited before, we would change the (x,y) value of our current cell to that of the neighbor, as well as set the corresponding wall to False, after which we would then recursively call our function with our new coordinate, visited list, and wall list. Once all cells had been visited, we assumed our visited/east/south lists would no longer satisfy our cases which called the program recursively, resulting in the function returning a tuple of the three lists of lists now changed (visited, east walls, south walls). Unfortunately, for reasons unbeknownst to us, the function instead returns the initial three "2D arrays". The Haskell code is near-finished and as far as we can tell should work, but it doesn't.

The Java Code

Once we had realized that we wanted to make a maze generator in Java as well we got started on that by thinking that we should use a 2D array of ints to represent cells and make each cell a wall. It used a DFS algorithm to go from cell to cell and changing them to halls as it went through. Doing it this way we discovered it had some problems, one of them was that because each cell had only one "wall" that the mazes it made didn't really make sense, they more looked like tunnels dug through the ground. It also didn't make a path to the end of the maze a lot of times because it was often visiting the same cell multiple times, causing the visited list to get full of duplicate cells. We unfortunately did not have time to fix this before our presentation.

For our final implementation we decided to start from scratch. We first had to change the way we thought about mazes. Instead of being a 2D array of ints, or cells with four walls, we figured out that all you needed to represent a maze was east walls, and south walls. To do this we just

made a 2D array of booleans for the eastern and southern walls. The cells where just the x and y coordinates of the data in the 2D arrays. When we visited a cell, we would make the corresponding cell in the wall array's False, depending on which direction we entered it. If we went west, we would make the eastern wall of the cell we are entering false, and for north we would do the same thing, but for south. Once we figured this out it became a lot easier to make. Our only problem was that it was difficult to recurse to the previously visited cells, for DFS you are supposed to go back to the previous cell visited, which we could do, but we could not then go the cell that was more previous to that one. To fix this problem we used a random number generator to change the current cell to a random cell, however this would make impossible mazes with halls that were impossible to get to. Then to fix this problem, instead of choosing a random cell, we chose a random cell that was already visited. Once we did this the program started generating solvable mazes of any size that we wanted. Java was nicer to write because we were more familiar with it and that made it much easier to picture the code in our head. The DFS algorithm also lends itself nicely to imperative languages, and was much easier to write because you could actually change things in the arrays without having to make a entirely new array. That being said Java also had its down sides. Some of our functions became long and hard to understand quickly because of how verbose Java is. It was also sometimes hard to debug, because most of the time it would work, but then every once in a while it would go into a infinite loop and we couldn't figure out why.

Project Architecture:

# of Lines:

daedalus.hs (first attempt) - 109
MazeGenerate.hs (final attempt) - 171
daedalus.java (first attempt) - 170
MazeGenerate.java (final attempt) - 143

Time Spent Coding(hours):

daedalus.hs (first attempt) - 24
MazeGenerate.hs (final attempt) - 10
daedalus.java (first attempt) - 10
MazeGenerate.java (final attempt) - 11

To see example outputs of our code, open outputs.txt.