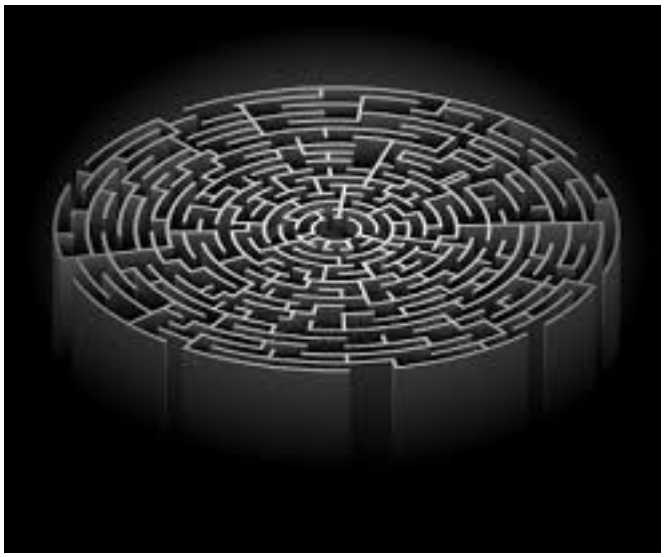
Generating Mazes in Idris

Felix Dilke

Generating mazes

- I've been trying to learn Idris, a programming language which is like Haskell but more so
- I (wrongly) decided I understood it well enough to try writing a program to generate mazes
- The simple things were hard, and the hard things were simple…



The mission, should you choose to accept it

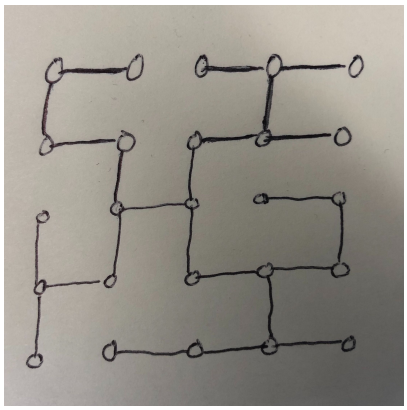
The idea is that the maze will look like this:

I also decided to output this in text mode using quarter-square graphics.

Once the basic algorithm is done, it can output bigger and badder mazes.

How do you generate a maze, anyway?

Abstracting away all the irrelevant details, the underlying skeleton of the maze is something like this:



This is a rectangular grid of cells where we've connected just enough pairs of adjacent cells for the whole graph to be one piece,

or, equivalently:

as many pairs of adjacent cells as we can without forming a circuit.

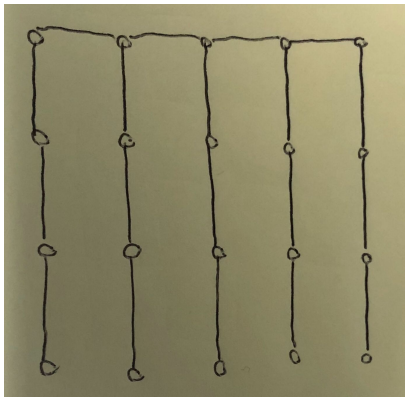
CompSci description of the problem

Given a rectangular grid of cells, make it into a graph by connecting every pair of adjacent cells with an edge.

We have to find a *spanning forest* of this graph.

This is pretty simple: you just accumulate a list of edges, only adding ones that don't create a circuit.

But, here's what happens if you do that



which is not an acceptable solution because it's a boring maze.

We have to introduce randomness, i.e. present the edges in a random order.

It turned out this was the hardest part of the project.

So I'm going to skate lightly over the actual Spanning Forest algorithm

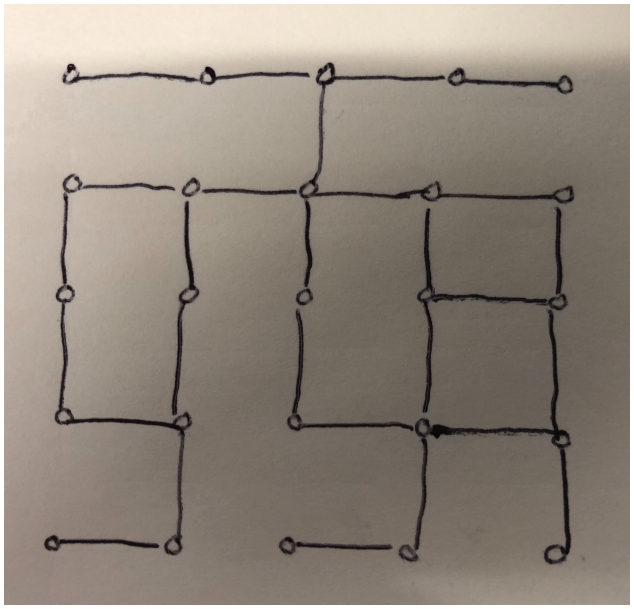
but here it is, anyway:

```
parameters (dset: SortedMap a a)
    export
    root: Eq a => a -> Maybe a
    root x = iterateToFixed fun (Just x) where
        fun: Maybe a -> Maybe a
        fun x = Lookup !x dset

    ||| Join two nodes in the context of a disjoint set, expressed as a (SortedMap a a)
    ||| Return an additional flag saying if the nodes were already joined
    export
    join: Eq a => a -> a -> (Bool, SortedMap a a)
    join x y = case (root x, root y) of
        (Nothing, Nothing) => (x == y, insert x y (insert y y dset))
        (Just xx, Just yy) =>
            if (xx == yy) then
                (True, dset)
            else
                (False, insert xx yy dset)
        (Just xx, Nothing) => (False, insert y xx dset)
        (Nothing, Just yy) => (False, insert x yy dset)

||| Find a spanning forest of a set of edges using Kruskal's algorithm
export
spanningForest: Ord a => List (a, a) -> List (a, a)
spanningForest edges =
    fst (foldr merge ([], fromList []) edges) where
        merge: (a, a) -> (List (a, a), SortedMap a a) -> (List (a, a), SortedMap a a)
        merge (x, y) (tree, adset) = case (join adset x y) of
            (True, _) => (tree, adset)
            (False, newadset) => ((x, y) :: tree, newadset)
```

This was hard too, but in a good way.

It's basically just a fold - accumulating a list of edges - but you have to efficiently keep track of which cells are connected. You also have to use the right data structures in Idris, and figure out how to test-drive it… all of which took a while.

But then, my mazes kept looking like this:



After staring at this for hours, I realized that

(1) the bottom and right edges were mangled because of a fencepost error, which was easily fixed

(2) the randomizer wasn't working properly.

But all it had to do was shuffle the list of edges, i.e. generate a uniformly random permutation.

Why is randomness so hard?

It isn't functional! A method that returns random numbers isn't permissible in Idris because it can return a different value on every call, violating the semantics. Same goes for a function that returns the date and time.

So you have to generate random numbers in the context of a special monad, and use the Effects library to get it to interoperate with all the other monads you have to use for anything that is not strictly functional.

Yes, I know this is why not everyone would want to use languages like Idris.

Also, it turned out there was a BUG in the run time library which was interacting adversely with the perhaps overcomplicated "Godel numbering scheme for permutations" I had decided to use, which was fun to implement but not, as it turned out, practical.

Why is randomness so hard? (continued)

Instead, the solution was to look up "generating uniformly random permutations" on Wikipedia which tells you to use the Knuth shuffle.

But even that is hard! The algorithm involves swapping successive pairs of elements in an array to generate the permutation by composing transpositions. And that's non-functional - arrays are not mutable in Idris.

I managed to write a recursive algorithm to do it by disassembling and recombining the array, but it was very slow for mazes of any size.

I searched the web to find out how Haskellers get round this problem, and it turns out they do it by using a special hack to mutate the array, because Haskell is a more mature (and possibly more pragmatic) language than Idris.

I conjecture that generating random permutations in an efficient yet functionally pure way should be achievable, but for now, this seems to be the stuff of CompSci PhD theses.

Meanwhile, here is my not-too-chronically-slow compromise solution:

```
shuffle : List a -> Eff (List a) [RND]
shuffle [] = pure []
shuffle [x] = pure [x]
shuffle (x::xs) = do
    shorter <- shuffle xs
    pos <- rndFin $ length xs
    let (left, right) = splitAt (toNat pos) shorter
    pure $ left ++ [x] ++ right
```

... which finally makes it possible ...

... to generate mazes in Idris

(switch to command line for demo)

THANK YOU