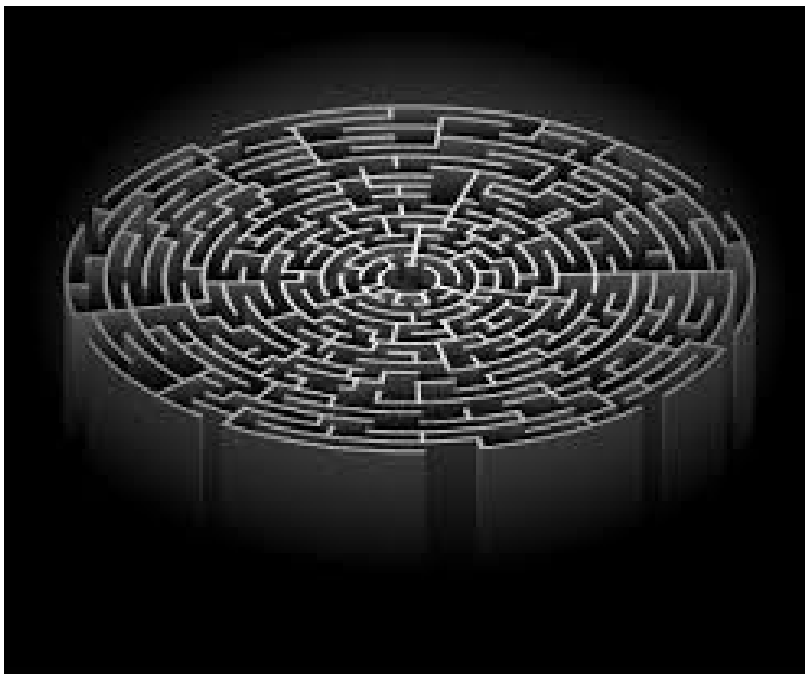
Generating Mazes in Idris

Felix Dilke

Mazes hold continuing fascination...

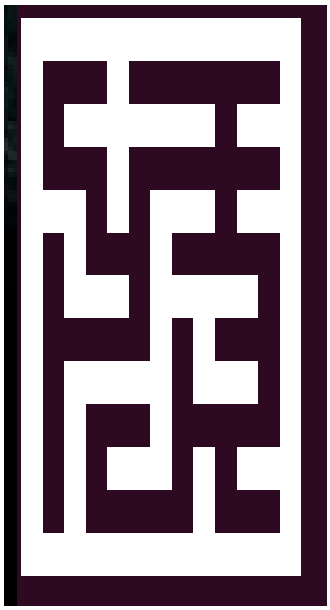


... so let's write a program to generate them

- I've been trying to learn Idris, a somewhat bleeding-edge 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 hard things turned out to be simple, but the simple things were hard...

The mission

How about a maze that looks like this:

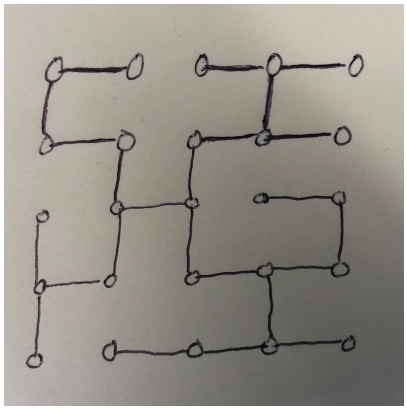


Let's output this in text mode using quarter-square graphics.

Once the basic algorithm is done, it can scale up to do bigger and badder mazes.

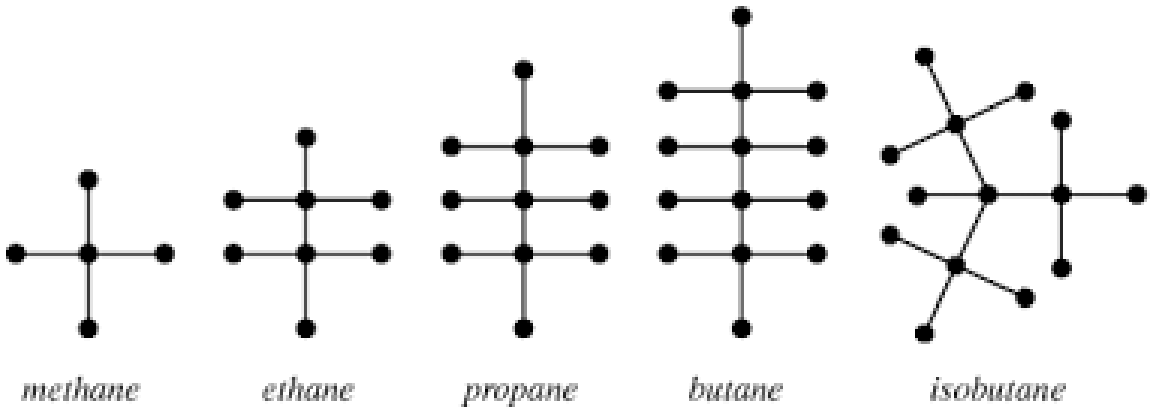
How do you generate a maze?

Abstracting away the irrelevant details, the structure of the maze is:



It's a rectangular grid of cells with connections added between adjacent cells to form a *tree*.

Trees - a concept from graph theory



A *tree* is a graph with just enough edges to make it one piece, or, equivalently, as many edges as possible without forming a circuit.

Poetically, a graph consisting of a bunch of trees is called a *forest*.

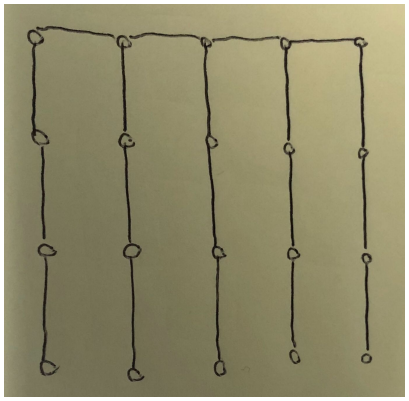
CompSci description of the problem

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 if you do that...



Not an acceptable solution: the same boring maze each time.

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

Absurdly, this was the hardest part of the project.

Skating lightly over the algorithm

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)
```

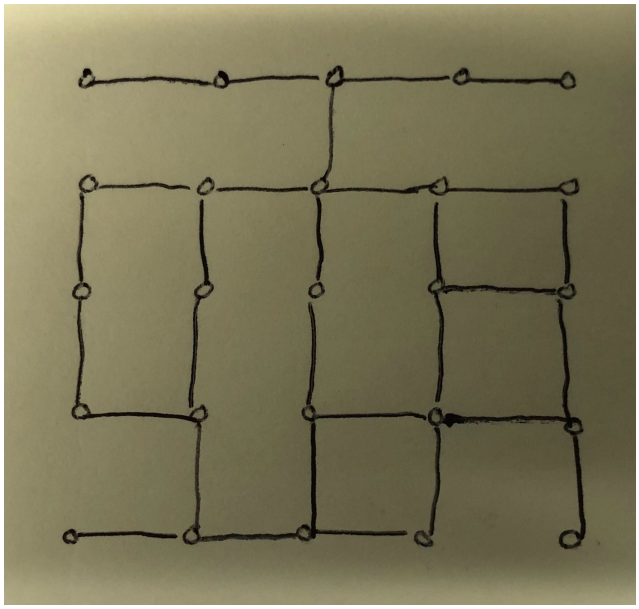
This was hard too, but in a good way.

Let me unpack that for you

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.

Debugging the mazes



After staring at this for hours, I realized that the bottom and right edges were mangled because of a fencepost error (easily fixed)

Also the randomizer wasn't working properly. But all it had to do was shuffle the list of edges…?

Why is randomness so hard?

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

Into the dark heart of the effects monad

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 else 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.

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.

Optimizing the shuffle

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.

An expedient hack

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) shor
    pure $ left ++ [x] ++ right
```

... which finally makes it possible ...

... to generate mazes in Idris

(switch to command line for demo)

THANK YOU