Code and Bugs

JUNE 6, 2011

As mentioned previously, I had an assignment which involved constructing a maze from code and letting a robot escape from it. This series won't follow the second part, it will concentrate on maze specific topics. I will write about maze generation, maze drawing, maze counting and several other aspects. Maybe I'll touch several properties of the mazes, possibly I'll write about something like the moments in statistics. Will see where this series will end and what areas will be covered by it.

This post will speak about maze construction. After a few searches for good and valid algorithms, I arrived at Jamis' blog. From there, after reading the exhaustive listing of generation procedures, I selected one algorithm to implement in Haskell and to present here. I choose the Sidewinder both because it has a nice name and because it seemed easy to implement in Haskell (requiring little state information and only a few random numbers). It runs very quickly and can be generated on the fly if needed.

It has two disadvantages though: the top row is free of walls and it can be easily solved. If starting on the lower row simple walk until you find an opening to the north and go in that direction. Repeat until you finish. If starting on the top row, you'll have to do a little backtracking but usually not a time-consuming one. Also, it doesn't produce all of the possible mazes. Solutions to all these problems will come in following articles, hopefully.

Now, for the generation part, I tweaked the algorithm a little. Instead of deciding at each cell whether to carve east or not, I started by deciding how many cells to carve east before doing any work. This will be clearer soon, after seeing the code.

First, some type definitions used to make the code look cleaner

```
type Length = Int
type Size = (Length, Length)
type Point = Size

{- The cardinal directions. -}
data Cardinal = N | E | S | W deriving (Eq, Show, Read, Ord, Enum)

{- A cell. The list contains the openings. -}
newtype Cell = C [Cardinal] deriving (Show, Read)

{- Simple type for maze. -}
type Maze = Array Size Cell
```

We have to generate a maze, we will use a function which receives the size of the maze as input and outputs the maze. Since we need randomness, we will be working in the State monad working with a random generator as the state (that's the `State StdGen` thing in the signature).

```
genMaze :: Size -> State StdGen Maze
```

Basically, the generation is split into two phases: one in which we generate the important points in the maze (ends for working sets and upwards openings) and one in which the actual building is done.

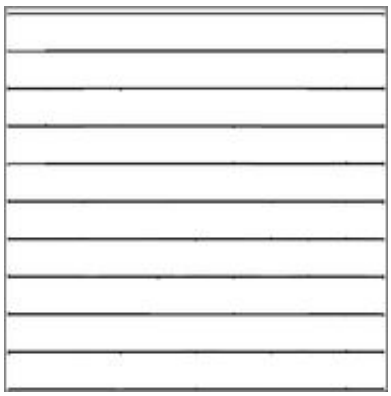
```
genMaze s@(sx, sy) = do
  (ews, ups) <- gMP s
  return $ build sx sy ews ups
```

The building process is very simple, it seems that it can be generalized to any other maze generation algorithm with only a few changes. However, this is not true, to use this process you'll have to obtain a list of important points and build the maze from it. It won't even work for on-the-fly generation. But this wasn't the point at that time and is not really relevant now.

```
build :: Length -> Length -> [Point] -> [Point] -> Maze
```

We will construct an array of cells, each of them opened to the East and West first.

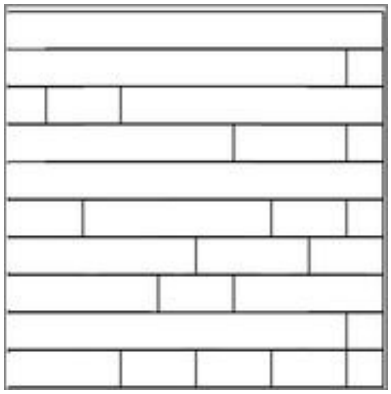
```
build sx sy ews ups = runSTArray $ do
  m <- newListArray ((1, 1), (sy, sx)) $ repeat $ C [E, W]
```



Only north and south walls

Then, using the first set of important points, the eastern openings of some cells will be closed. Using the same set, we will also close some of the western openings such that if a cell claims to be closed eastward the next cell will be closed westward.

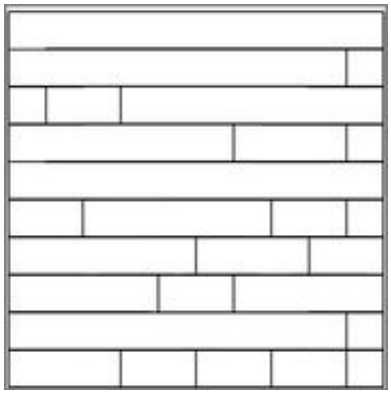
```
  mapM_ (blockCell m E) $ (sx, 1) : ews
  mapM_ (blockCell m W . first (+1)) $ filter (fst . first (/
= sx)) ews
```



East and West closed

After blocking the western edges of each row we will have a set of rows, like in the following picture

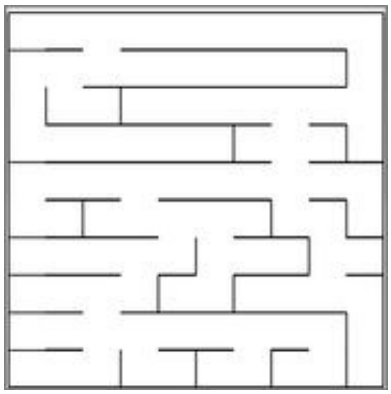
```
mapM_ (blockCell m W . (\y -> (1, y))) [1 .. sy]
```



Rows fully closed

We'll have to open some cells northwards and some southwards, just like we did with the East-West relationship. Then, we will have our maze.

```
mapM_ (openCell m N) ups
mapM_ (openCell m S . second (subtract 1)) ups
return m
```



Full maze

Opening and closing cells is in fact modifying the contents of the array.

```
{-
Block one cell from the maze, represented as an array.
-}
-- blockCell :: Data.Array.MArray Size Cell -> Cardinal -> Size -> m ()
blockCell m d (x, y) = do
  e <- readArray m (y, x)
  writeArray m (y, x) $ block e d

{-
Open one cell from the maze, represented as an array.
-}
-- openCell :: (MArray a Cell m) => a Size Cell -> Cardinal -> Size -> m ()
openCell m d (x, y) = do
  e <- readArray m (y, x)
  writeArray m (y, x) $ open e d

{- Block a cell from one direction. -}
block :: Cell -> Cardinal -> Cell
block (C l) x = C $ filter (/= x) l

{- Open a cell to one direction. -}
open :: Cell -> Cardinal -> Cell
open (C l) x = C $ if x `elem` l then l else x : l
```

The only remaining part is the generation of the important points, as given by the Sidewinder's algorithm. Since the first row is special (no walls blocking eastward) we will skip it when generating the points. Only the other rows will be used to generate the important Row Points. After this, do a simple fold to construct the Maze Points used in the above build procedure.

```
gMP :: Size -> State StdGen ([Point], [Point])
gMP (sx, sy) = do
  points <- mapM (gRP 0 sx) [2..sy]
  return $ foldl (\(x, y) (a, b) -> (x ++ a, y ++ b)) ([], []) points
```

Generating the RowPoints is simple: generate two random numbers and return the points obtained from them. More exactly, construct a list of them. The numbers will represent the length of the current corridor and the offset at which the opening to the previous row will be carved.

```
gRP :: Coord -> Length -> Coord -> State StdGen ([Point], [Point])
gRP c sx y
  | sx <= 0 = return ([], [])
  | otherwise = do
    len <- state $ randomR (1, sx)
    up <- state $ randomR (1, len)
    (rx, ry) <- gRP (c + len) (sx - len) y
    return ((len + c, y):rx, (up + c, y):ry)
```

Basically, the above function is the only one which needs random numbers. However, since we want to be able to change the generation if another, more efficient, algorithm is found, the only exported function should be `genMaze`.

The entire code can be seen on Github.

Next time we will look at how to do the actual rendering of this maze in Haskell (the above pictures were generated using a modified version of what will be presented then). Afterwards, we'll touch some more interesting topics related to mazes but I don't want to spoil anything right now.

Original URL:

http://pgraycode.wordpress.com/2011/06/06/mazes-1/