Maze Generator and Solver in Clojure

Jack Corton b6005726

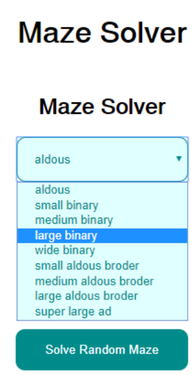
Key Data Structures and Concepts

Maze Structure

Each cell was represented as Clojure map with the keys North, East, South and West with a value of 0 for a wall in that direction and a 1 for no wall. Each cell map was then held in a vector with other cells to represent a row and the rows were then held in a final vector. Once Dijkstra's algorithm is implemented they have a distance appended under *:count*.

```
[[{:north 0, :east 1, :south 0, :west 0}
  {:north 0, :east 1, :south 1, :west 1}
  {:north 0, :east 0, :south 0, :west 1}]
 [{:north 0, :east 0, :south 1, :west 0}
  {:north 1, :east 0, :south 1, :west 0}
  {:north 0, :east 0, :south 1, :west 0}]
 [{:north 1, :east 1, :south 0, :west 0}
  {:north 1, :east 1, :south 0, :west 1}
  {:north 1, :east 0, :south 0, :west 1}]]
```

```
[[{:north 0, :east 1, :south 0, :west 0, :count 0}
  {:north 0, :east 1, :south 1, :west 1, :count 1}
  {:north 0, :east 0, :south 0, :west 1, :count 2}]
 [{:north 0, :east 0, :south 1, :west 0, :count 5}
  {:north 1, :east 0, :south 1, :west 0, :count 2}
  {:north 0, :east 0, :south 1, :west 0, :count 5}]
 [{:north 1, :east 1, :south 0, :west 0, :count 4}
  {:north 1, :east 1, :south 0, :west 1, :count 3}
  {:north 1, :east 0, :south 0, :west 1, :count 4}]]
```



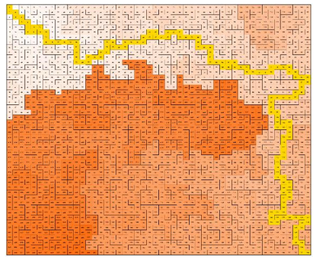
Web Page Structure

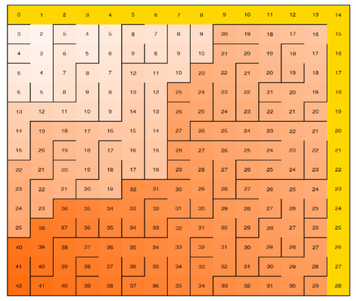
Using Hiccup (creating HTML elements in Clojure), Compojure (a routing library) and Ring (HTTP server library) the solved maze was able to be represented on-screen (using a HTML table) and mazes in the NoSQL Mongo database were able to be retrieved using a MongoDB library called Monger. This created a page with a dynamically populated drop down of mazes, a create a maze form for both algorithms, solve a non-stored, newly generated maze and a random maze from the DB button.

Algorithms Used

Dijkstra's Algorithm

Dijkstra's algorithm measures the shortest distance between a starting point (which we specify), and every other cell in the maze.
It works by starting at the start point then finding its available neighbours, then finding the available neighbours of each neighbour, using a queue, until the mazes is 'flooded' (Buck, 2015) with the distances from the start.



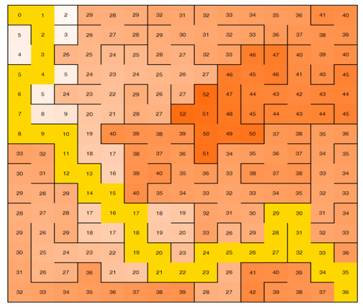


Binary Tree Algorithm

This is a very simple algorithm that chooses between carving east or north on each cell and through doing this on every cell it will soon carve a maze (Buck, 2015). It can only carve east on the top row and north on the eastern row which creates a long corridor on each maze, and it is biased towards diagonal routes towards the bottom left corner. Although the simplest maze algorithm and possibly the least performance taxing it does not produce a perfect set of maze routes.

Aldous-Broder Algorithm

The idea of this maze algorithm is to start on a random cell and carve in any direction (in the grids boundaries) and keep visiting random neighbours, if it's a new cell then carve to it and if not just visit any neighbour of the new cell. This is a random walk and creates unbiased mazes (Buck, 2015). It can take a while to run as it could take a time to randomly walk to the last few remaining cells but the trade off for an unbiased maze is quite worth it.



Testing

For the mazes I tested using a console print function to check visually whether they came out correctly. This allowed me to easily spot issues such as earlier mazes carving walls on the outside of the maze where they should not be doing that. As simple as it sounds, using pen and paper for the smaller mazes and verifying the shortest path and distances.

It generated binary mazes with small (10x10), medium (25x25) and large(50x50) which all worked fine. Even 300 by 300 worked using Aldous Broder, albeit it took a while to display being such a large maze.

References

References

Buck, J. (2015). *Mazes for Programmers*. Pragmatic Bookshelf.

How Clojure Was Used and Contrasts With OOP

How Clojure was used – Maze Gen

To alter grid cells the maze was returned with the assoc-in function which returned a maze with the appropriate cells changed. This meant comparing maze to see if a passage was carved (Aldous Broder) was simple as using `(= maze new-maze)`.

Using clojure/data.json was imperative to each of the projects as using a map to JSON function was needed to store the mazes in the database, using Monger, as JSONs then once retrieved in the client they could be translated back to a Clojure structure which would be ready to solve.

```
:else
(if (= 0 (rand-int 2))   ;randomly carve north or
  (assoc-in (assoc-in grid [x (+ y 1) :west] 1)
  (assoc-in (assoc-in grid [(- x 1) y :south] 1
```

name String	maze String
"aldous"	"[[{"north":0,"east":1,"south":
"small binary"	"[[{"north":0,"east":1,"south":
"medium binary"	"[[{"north":0,"east":1,"south":
"large binary"	"[[{"north":0,"east":1,"south":
"wide binary"	"[[{"north":0,"east":1,"south":
"small aldous broder"	"[[{"north":0,"east":0,"south":
"medium aldous broder"	"[[{"north":0,"east":1,"south":
"large aldous broder"	"[[{"north":0,"east":1,"south":
"super large ad"	"[[{"north":1,"east":1,"south":

```
[:table {:style "border-collapse: collapse; margin:20px auto"}
 (for [row maze]
   [:tr (for [col row]
          [:td
           {:style (str "width:60px; height:60px; text-align:center; fo
                        ;if the cell exists on the shortest path then c
                        ;if not colour normally
                        (if (some true? (map #(if (and (= (% :x) (col :x
                          "background: rgba(255,215,0, 1);"
                          (str "background: rgba(255, 110, 17," (str (
                        ;Set borders for walls in the maze
                        (if (= (col :north) 0) "border-top: 2px solid b
                        (if (= (col :east) 0) "border-right: 2px solid
                        (if (= (col :south) 0) "border-bottom: 2px soli
                        (if (= (col :west) 0) "border-left: 2px solid b
           (if (contains? col :count) (str (col :count)) "X")])])])
```

How Clojure was used – Solver

In the client maze solver, Clojure was used similarly to the maze generator server. The mazes were solved using many functions for the many elements that form Dijkstras algorithm which translated well from pseudo code.

The solver relies on Hiccup to create web pages and the maze onscreen as a HTML table. The maze is created by creating table rows <tr> and table cells <td> in the same shape as the maze and then adding borders in CSS to represent passages. Then the cells are given a colour intensity of its distance divided the largest distance in the maze unless its on the shortest path then it is given a different colour to show that.

Implementation in Clojure

Clojure had many benefits while developing the server/client maze solver.

- Tight integration with the Java VM means
- Clean syntax due to being a LISP derivative

Using recursion instead of iterative looping means cleaner code, keeping immutability on variables instead of using a Clojure Atom and keeping idiomatic code with functions that can be reused in many different scenarios.

Using functions to return values rather than assigning them to locally scoped mutable variables is a much more intricate way to use data as each function can manipulate parameters to return different values.

Implementation in OOP

In OOP the algorithms would've used locally scoped variables that are mutable and could have their valued changed instead of recurring and passing through a modified version of said variable.

Would have looped using while/for instead of using recursion once again due to mutability meaning values can be changed and not having to pass through values and mapping them to local immutable variables.

OOP would also benefit from using classes and constructors to create the base maze e.g. being able to use something similar to `maze = new maze(10,10)` to create a blank maze.