

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*.

Implementation in Clojure

* Used recursion and brought in variables as slightly modified versions of variables before in the maze algorithms. Being able to
* Had immutability on the variables meaning they could not be altered, only altered through passing a modified version through a recursion
* Using functions to return values back makes it very simple to structure an algorithm

Implementation in OOP

* Would have used a 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 instead of using recursion once again due to mutability and not having to pass through values and mapping them to local immutable variables
* Using classes and in-class functions/constructors to create the base maze such as being able to use something similar to maze = new maze(10,10).

How Clojure Was Used and Contrasts With OOP

References

# References

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

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 from the start on every

Algorithms Used

Dijkstra’s Algorithm

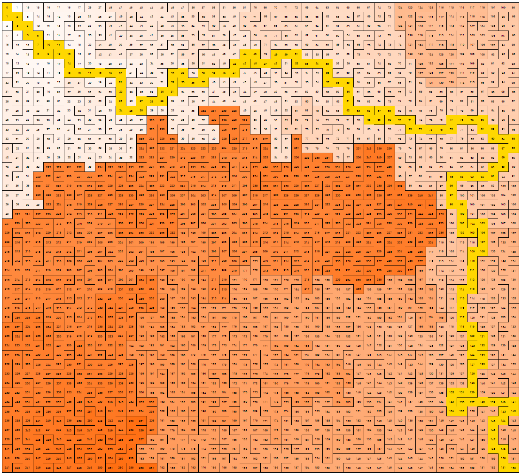
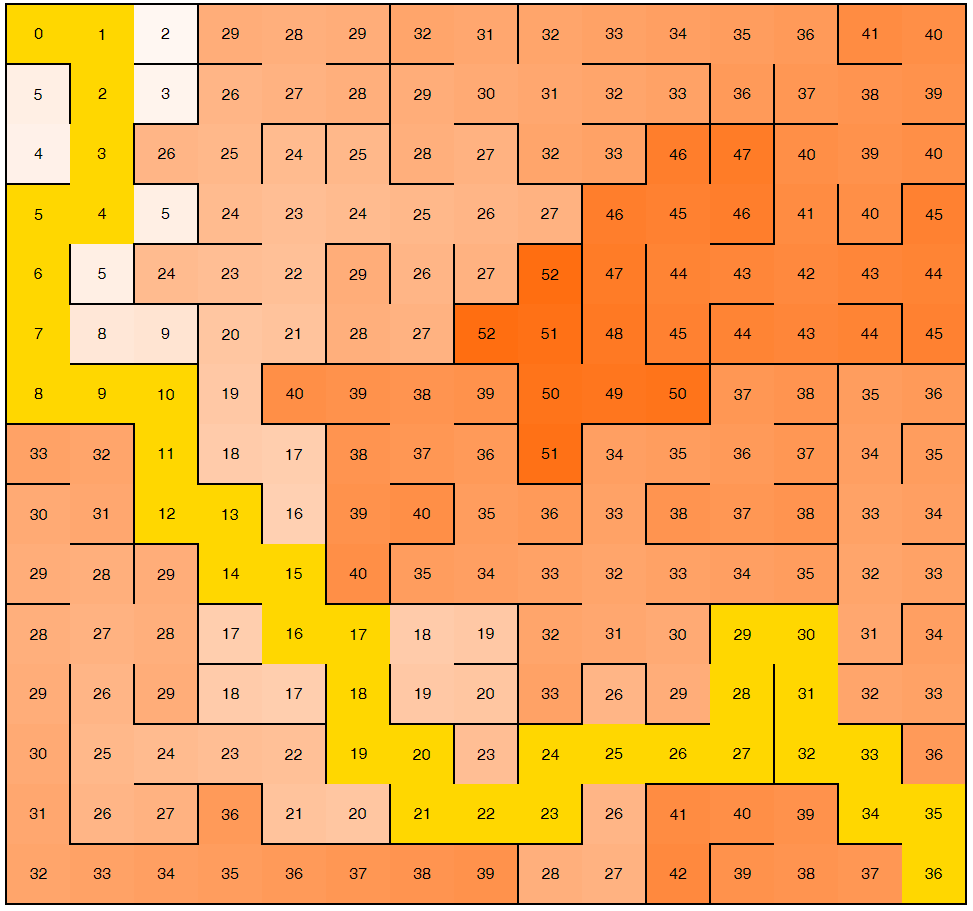
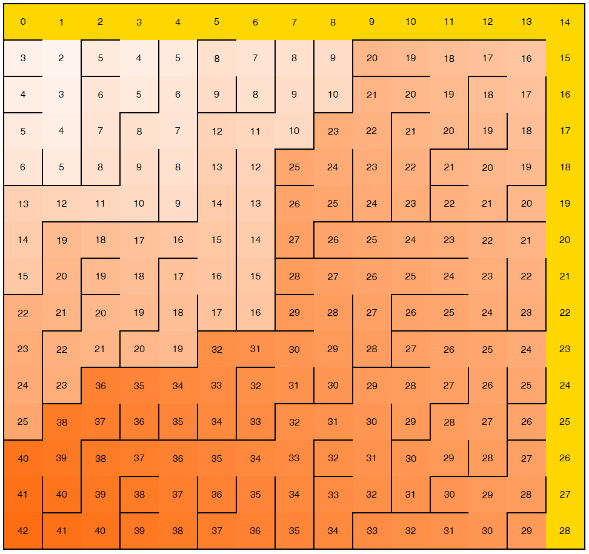
Dijkstra’s algorithm measures the shortest distance between some starting point (which we specify), and every other cell in the maze.   
It works by starting at the start point then finding its available neighbours, then finding the available neighbours of each neighbour until the mazes is ‘flooded’ 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.



How Clojure was used

* Understanding of the language is demonstrated
* Its relevant merits are shown
* Difficulties using it are identified
* Criteria are set and provide a reasonable basis on which to build judgements
* Conclusions are based on firm data

Key Data Structures and Concepts

Maze Generator and Solver in Clojure  
Jack Corton b6005726



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 list of mazes and create a maze form.