Maze Generation

Submitted by

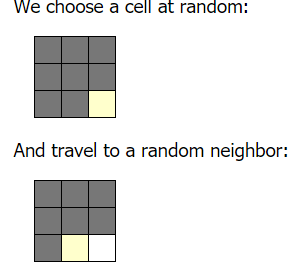
Muhammad Alvin Pramadrian

WID180708

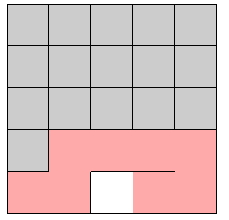
# **Algorithm**

## **Recursive Backtracker**

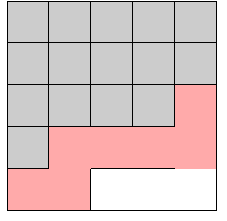
This Algorithm Starts by selecting a random tile, and starts to traverse from there



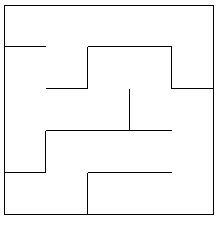
Each time we travel through a neighbor, we add them to a passaged tile. This repeats until the algorithm finds a blocked path



Once it passed a fully Initialized tile, the algorithm will trace back to the last tile where the tile isn’t fully Initialized and continues from there



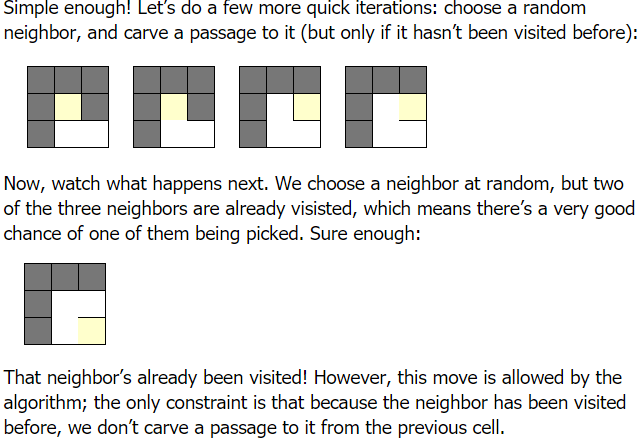
Final Result



|  |
| --- |
| def carve\_passages\_from(cx, cy, grid) |
|  | directions = [N, S, E, W].sort\_by{rand} |
|  |  |
|  | directions.each do |direction| |
|  | nx, ny = cx + DX[direction], cy + DY[direction] |
|  |  |
|  | if ny.between?(0, grid.length-1) && nx.between?(0, grid[ny].length-1) && grid[ny][nx] == 0 |
|  | grid[cy][cx] |= direction |
|  | grid[ny][nx] |= OPPOSITE[direction] |
|  | carve\_passages\_from(nx, ny, grid) |
|  | end |
|  | end |
|  | end |

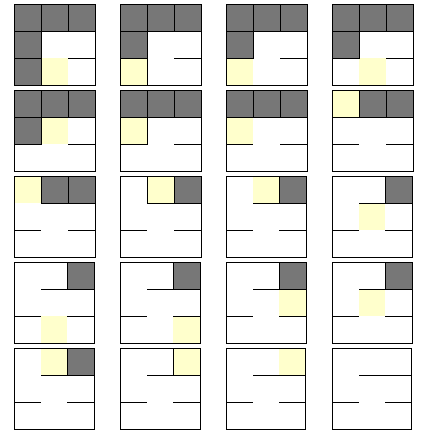
## **Aldous Broder Algorithm**

The Algorithm works almost the same as recursive backtracking, the difference being



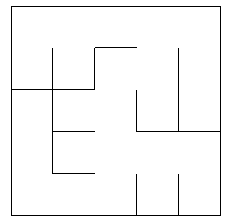
Instead of recursing back, it allows the algorithm to pass through a “passed” tile. It continues

Until the maze is complete



|  |
| --- |
| x, y = rand(width), rand(height) |
|  | remaining = width \* height - 1 |
|  |  |
|  | while remaining > 0 |
|  | display\_maze(grid, x, y) |
|  | sleep 0.02 |
|  |  |
|  | [N,S,E,W].shuffle.each do |dir| |
|  | nx, ny = x + DX[dir], y + DY[dir] |
|  | if nx >= 0 && ny >= 0 && nx < width && ny < height |
|  | if grid[ny][nx] == 0 |
|  | grid[y][x] |= dir |
|  | grid[ny][nx] |= OPPOSITE[dir] |
|  | remaining -= 1 |
|  | end |
|  |  |
|  | x, y = nx, ny |
|  | break |
|  | end |
|  | end |
|  | End |

Maze result:



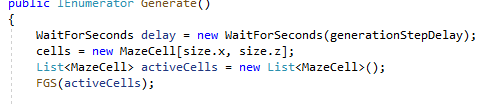
# **IMPLEMENTATION**

For this project, we’ll use the idea of both **Recursive** and **Aldous** Algorithm

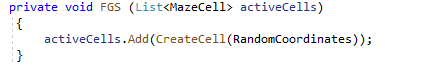
Each cell will be represented by a tile



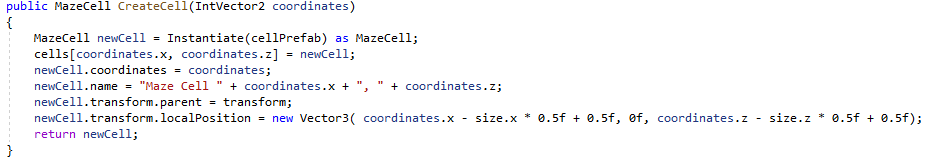
To generate the Maze, we Initialized the size of the maze, and create a list to represent the passed cell and do a randomized first tile called “*First Generation Step”*



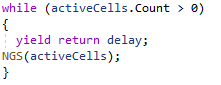
FGS:



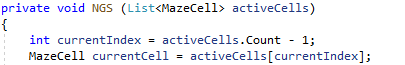
Create Cell:



As the FGS add a cell to the *activeCells* list, we continued to generate the rest of the maze



The NGS would take the last element of the List, or so I say would act as a Stack in this case



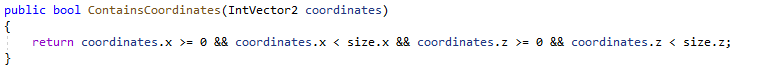
Next, take a random direction and add it to the current cell’s position



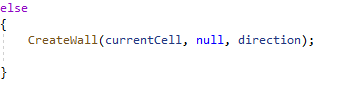
Next, we check if the new coordinate is inside the cell



This method will check if the next coordinate is inside the Maze



If no, then it will create a wall to create the borders of the Maze



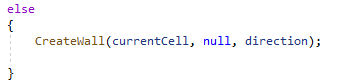
If it is inside, we will fetch the cell of the coordinate



If the neighbor is empty, then it will create a cell and add it to the *activeCells* List/Stack

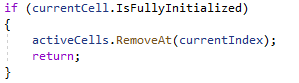


If the neighbor is activated already, it will instead create a wall



As it Is a while loop, the Algorithm will repeat itself until the *activeCell* is empty, this replicates the **Aldous** algorithm

If the last element *isFullyInitialized* (Ie, all 4 directions of North, East, South, and West are not empty), then it will remove that element and break the loop



This would replicate the **Recursive Backtracking** whereas it’s not recursive, but it IS backtracking by representing the recursion as a List

# **Maze Result:**

