Coding Example: Find shortest path in a maze

In this lesson, we will look how to find the shortest path in a maze.

WE'LL COVER THE FOLLOWING

Problem Description

Path finding is all about finding the shortest path in a graph. This can be split in two distinct problems:

- To find a path between two nodes in a graph
- To find the shortest path.

We'll illustrate this through path finding in a maze. The first task is thus to build a maze!

A hedge maze at Longleat stately home in England. Image by `Prince Rurik (https://commons.wikimedia.org/wiki/File:Longleat_maze.jpg), 2005.

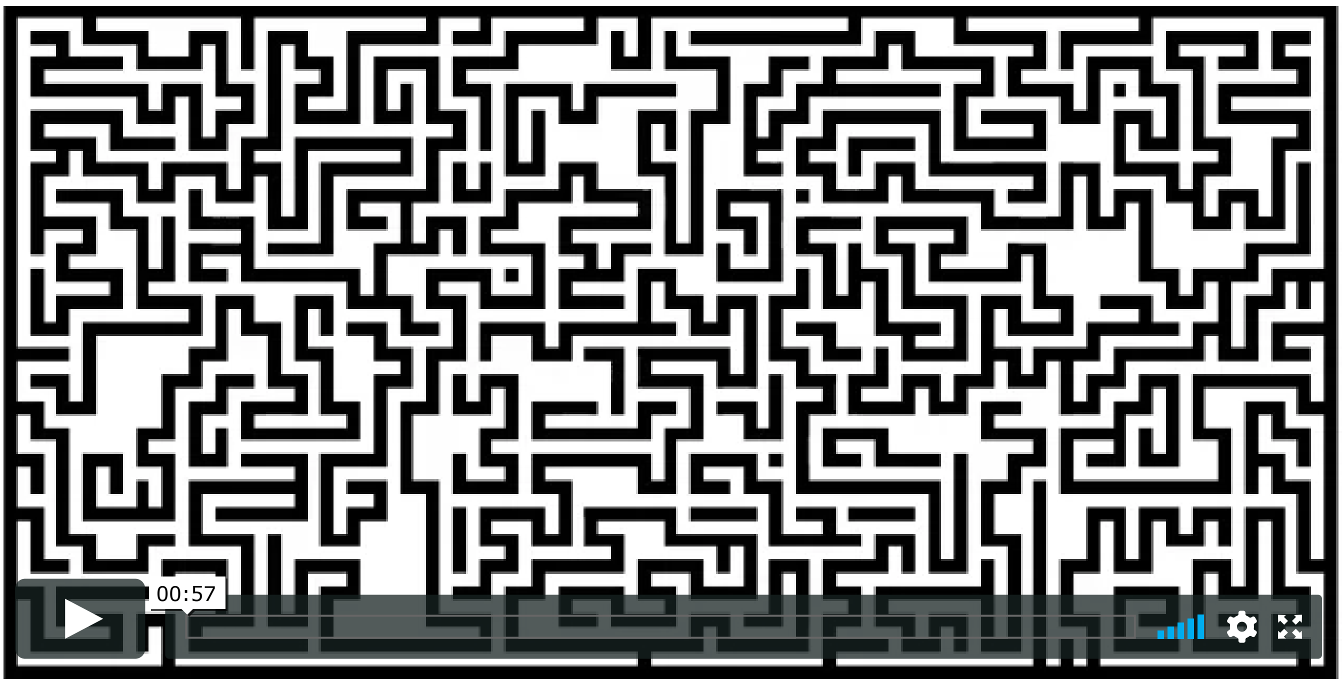
Building a maze

There exist many maze generation algorithms but I tend to prefer the one I've been using for several years but whose origin is unknown to me. I've added the code in the cited Wikipedia entry. Feel free to complete it if you know the original author.

This algorithm works by creating `n` (density) islands of length `p` (complexity). An island is created by choosing a random starting point with odd coordinates, then a random direction is chosen. If the cell two steps away in the given direction is free, then a wall is added at both one and two steps in this direction. The process is iterated for `n` steps for this island. `p` islands are created. `n` and `p` are expressed as a float to adapt them to the size of the maze. With a low complexity, islands are very small and the maze is easy to solve. With a low density, the maze has more "big empty rooms".

```
# -----------------------------------------------------------------------------
# From Numpy to Python
# Copyright (2017) Nicolas P. Rougier - BSD license
# More information at https://github.com/rougier/numpy-book
# -----------------------------------------------------------------------------
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation



def build_maze(shape=(65, 65), complexity=0.75, density=0.50):
    """
    Build a maze using given complexity and density

    Parameters
    ==========

    shape : (rows,cols)
      Size of the maze

    complexity: float
      Mean length of islands (as a ratio of maze size)

    density: float
      Mean numbers of highland (as a ratio of maze surface)

    """

    # Only odd shapes
    shape = ((shape[0]//2)*2+1, (shape[1]//2)*2+1)

    # Adjust complexity and density relatively to maze size
    n_complexity = int(complexity*(shape[0]+shape[1]))
    n_density = int(density*(shape[0]*shape[1]))


    # Build actual maze
    Z = np.zeros(shape, dtype=bool)

    # Fill borders
    Z[0, :] = Z[-1, :] = Z[:, 0] = Z[:, -1] = 1

    # Islands starting point with a bias in favor of border
    P = np.random.normal(0, 0.5, (n_density, 2))
    P = 0.5 - np.maximum(-0.5, np.minimum(P, +0.5))
    P = (P*[shape[1], shape[0]]).astype(int)
    P = 2*(P//2)

    # Create islands
    for i in range(n_density):

        # Test for early stop: if all starting point are busy, this means we
        # won't be able to connect any island, so we stop.
        T = Z[2:-2:2, 2:-2:2]
        if T.sum() == T.size:
            break

        x, y = P[i]
        Z[y, x] = 1
        for j in range(n_complexity):
            neighbours = []
            if x > 1:
                neighbours.append([(y, x-1), (y, x-2)])
            if x < shape[1]-2:
                neighbours.append([(y, x+1), (y, x+2)])
            if y > 1:
                neighbours.append([(y-1, x), (y-2, x)])
            if y < shape[0]-2:
                neighbours.append([(y+1, x), (y+2, x)])
            if len(neighbours):
                choice = np.random.randint(len(neighbours))
                next_1, next_2 = neighbours[choice]
                if Z[next_2] == 0:
                    Z[next_1] = 1
                    yield Z.copy()
                    Z[next_2] = 1
                    yield Z.copy()
                    y, x = next_2
            else:
                break
    return Z


if __name__ == '__main__':
    S = []
    for i, Z in enumerate(build_maze((51, 101))):
        S.append(Z)

    fig = plt.figure(figsize=(10, 5))
    ax = fig.add_axes([0.0, 0.0, 1.0, 1.0], frameon=False, aspect=1)
    im = ax.imshow(S[0], cmap=plt.cm.gray_r,
                   interpolation='nearest', animated=True)
    ax.set_xticks([]), ax.set_yticks([])
    Writer = animation.writers['ffmpeg']
    writer = Writer(fps=15, metadata=dict(artist='Me'), bitrate=1800)
    def update(frame):
        im.set_array(S[frame])
        return im,

    anim = animation.FuncAnimation(fig, update, interval=10, frames=100)
    anim.save('output/output.mp4', writer=writer)
    plt.show()
```

Here is an animation showing the generation process:



Progressive maze building with complexity and density control

Next, we will try to come up with a solution to find shortest path to solve this maze puzzle using the Breadth-First approach!