Coding Example: Find shortest path in a maze (Breadth-First approach)

In this lesson, we will implement the solution of finding shortest path in the maze that we build in the previous lesson. To find the shortest path, we will use the BFS strategy.

WE'LL COVER THE FOLLOWING

Breadth-First Algorithm

The breadth-first (as well as depth-first) search algorithm addresses the problem of finding a path between two nodes by examining all possibilities starting from the root node and stopping as soon as a solution has been found (destination node has been reached). This algorithm runs in linear time with complexity in $O(|V| + |E|)$ (where V is the number of vertices, and E is the number of edges).

Writing such an algorithm is not especially difficult, provided you have the right data structure. In our case, the array representation of the maze is not the most well-suited and we need to transform it into an actual graph as proposed by Valentin Bryukhanov.

Step 1: Implement a Graph Class

```
def build_graph(maze):
  height, width = maze.shape
  graph = {(i, j): [] for j in range(width)
                      for i in range(height) if not maze[i][j]}
  for row, col in graph.keys():
    if row < height - 1 and not maze[row + 1][col]:
      graph[(row, col)].append(("S", (row + 1, col)))

      graph[(row + 1, col)].append(("N", (row, col)))
    if col < width - 1 and not maze[row][col + 1]:
      graph[(row, col)].append(("E", (row, col + 1)))
      graph[(row, col + 1)].append(("W", (row, col)))
  return graph
```

> **Note:** If we had used the depth-first algorithm, there is no guarantee to find the shortest path, only to find a path (if it exists).

Step 2: Implement Breadth-First Algorithm

Once this is done, writing the breadth-first algorithm is straightforward. We start from the starting node and we visit nodes at the current depth only (breadth-first, remember?) and we iterate the process until reaching the final node, if possible. The question is then: do we get the shortest path exploring the graph this way? In this specific case, "yes", because we don't have an edge-weighted graph, i.e. all the edges have the same weight (or cost).

```
def breadth_first(maze, start, goal):
  queue = deque([([start], start)])
  visited = set()
  graph = build_graph(maze)

  while queue:
    path, current = queue.popleft()
    if current == goal:
      return np.array(path)
    if current in visited:
      continue
    visited.add(current)
    for direction, neighbour in graph[current]:
      p = list(path)
      p.append(neighbour)
      queue.append((p, neighbour))
  return None
```

Complete Solution

Let's merge all this logic into one code and visualize the maze, gradient, and the shortest path! Run the following code and once the output is generated, zoom-in to have a clearer view at the shortest path.

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


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] = Z[next_2] = 1
                    y, x = next_2
            else:
                break
    return Z


# ------------------------------------------------------ find_shortest_path ---

def build_graph(maze):
    height, width = maze.shape
    graph = {(i, j): [] for j in range(width) for i in range(height) if not maze[i][j]}
    for row, col in graph.keys():
        if row < height - 1 and not maze[row + 1][col]:
            graph[(row, col)].append(("S", (row + 1, col)))
            graph[(row + 1, col)].append(("N", (row, col)))
        if col < width - 1 and not maze[row][col + 1]:
            graph[(row, col)].append(("E", (row, col + 1)))
            graph[(row, col + 1)].append(("W", (row, col)))
    return graph

def BreadthFirst(maze, start, goal):
    queue = deque([([start], start)])
    visited = set()
    graph = build_graph(maze)
    while queue:
        path, current = queue.popleft()
        if current == goal:
            return np.array(path)
        if current in visited:
            continue
        visited.add(current)
        for direction, neighbour in graph[current]:
            p = list(path)
            p.append(neighbour)
            queue.append((p, neighbour))
    return None


# -------------------------------------------------------------------- main ---
if __name__ == '__main__':

    Z = build_maze((41,81))
    start, goal = (1,1), (Z.shape[0]-2, Z.shape[1]-2)

    P = BreadthFirst(Z, start, goal)
    X, Y = P[:,1], P[:,0]

    # Visualization maze, gradient and shortest path
    plt.figure(figsize=(13, 13*Z.shape[0]/Z.shape[1]))
    ax = plt.subplot(1, 1, 1, frameon=False)
    ax.imshow(Z, interpolation='nearest', cmap=plt.cm.gray_r, vmin=0.0, vmax=1.0)
    cmap = plt.cm.hot
    cmap.set_under(color='k', alpha=0.0)
    ax.scatter(X[1:-1], Y[1:-1], s=60,
               lw=1, marker='o', edgecolors='k', facecolors='w')
    ax.scatter(X[[0,-1]], Y[[0,-1]], s=60,
               lw=3, marker='x', color=['w','k'])
    ax.set_xticks([])
    ax.set_yticks([])

    plt.tight_layout()
    plt.savefig("output/maze.png")
    plt.show()
```

In the next lesson, we will try to solve this same problem using the Bellman-Ford approach!