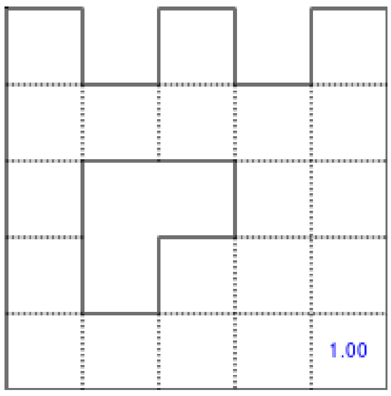
Coding Example: Find shortest path in a maze (Bellman-Ford approach)

In this lesson, we will implement the solution of finding the shortest path in the maze using the Bellman-Ford approach.

Bellman-Ford Algorithm

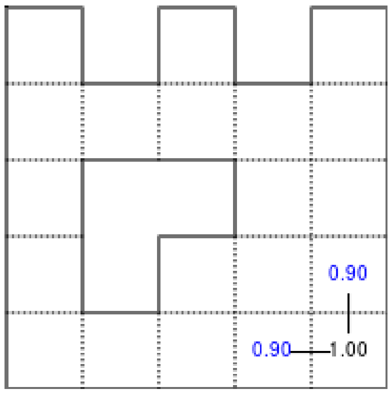
The Bellman–Ford algorithm is an algorithm that is able to find the optimal path in a graph using a diffusion process. The optimal path is found by ascending the resulting gradient. This algorithm runs in quadratic time $O(|V||E|)$ (where V is the number of vertices, and E is the number of edges).

However, in our simple case, we won't hit the worst case scenario. Once this is done, we can ascend the gradient from the starting node. You can check on the figure that this leads to the shortest path.



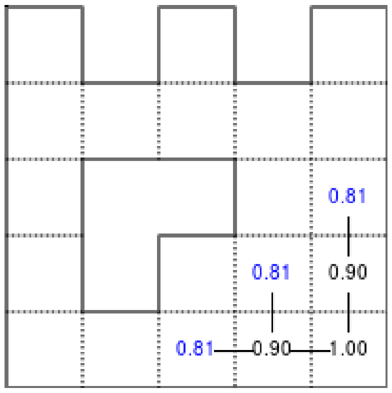
Value iteration algorithm on a simple maze. Once entrance has been reached, it is easy to find the shortest path by ascending the value gradient.

1 of 10



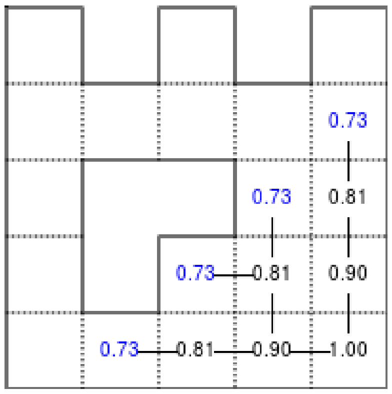
Value iteration algorithm on a simple maze. Once entrance has been reached, it is easy to find the shortest path by ascending the value gradient.

2 of 10



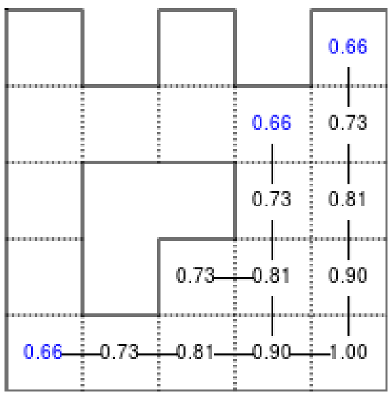
Value iteration algorithm on a simple maze. Once entrance has been reached, it is easy to find the shortest path by ascending the value gradient.

3 of 10



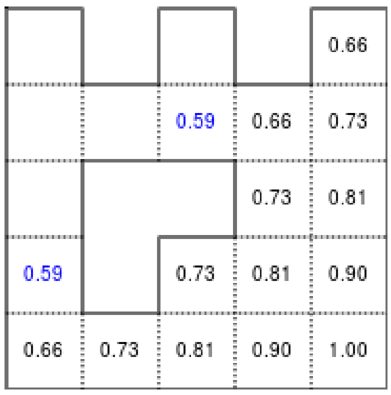
Value iteration algorithm on a simple maze. Once entrance has been reached, it is easy to find the shortest path by ascending the value gradient.

4 of 10



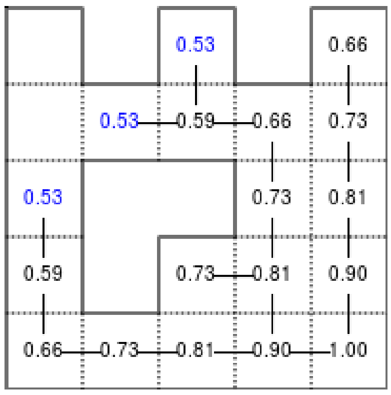
Value iteration algorithm on a simple maze. Once entrance has been reached, it is easy to find the shortest path by ascending the value gradient.

5 of 10



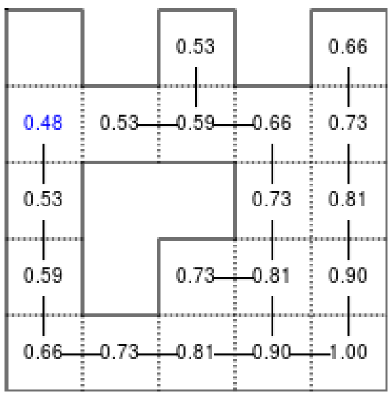
Value iteration algorithm on a simple maze. Once entrance has been reached, it is easy to find the shortest path by ascending the value gradient.

6 of 10



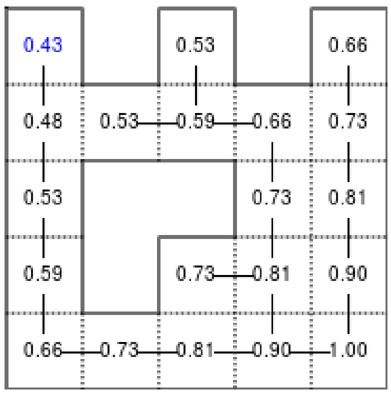
Value iteration algorithm on a simple maze. Once entrance has been reached, it is easy to find the shortest path by ascending the value gradient.

7 of 10



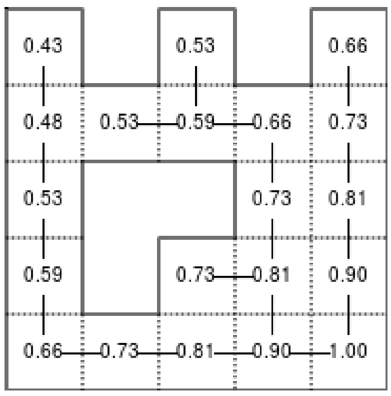
Value iteration algorithm on a simple maze. Once entrance has been reached, it is easy to find the shortest path by ascending the value gradient.

8 of 10



Value iteration algorithm on a simple maze. Once entrance has been reached, it is easy to find the shortest path by ascending the value gradient.

9 of 10



Value iteration algorithm on a simple maze. Once entrance has been reached, it is easy to find the shortest path by ascending the value gradient.

10 of 10

We start by setting the exit node to the value 1, while every other node is set to 0, except the walls. Then we iterate a process such that each cell's new value is

computed as the maximum value between the current cell value and the discounted (`gamma=0.9` in the case below) 4 neighbour values. The process starts as soon as the starting node value becomes strictly positive.

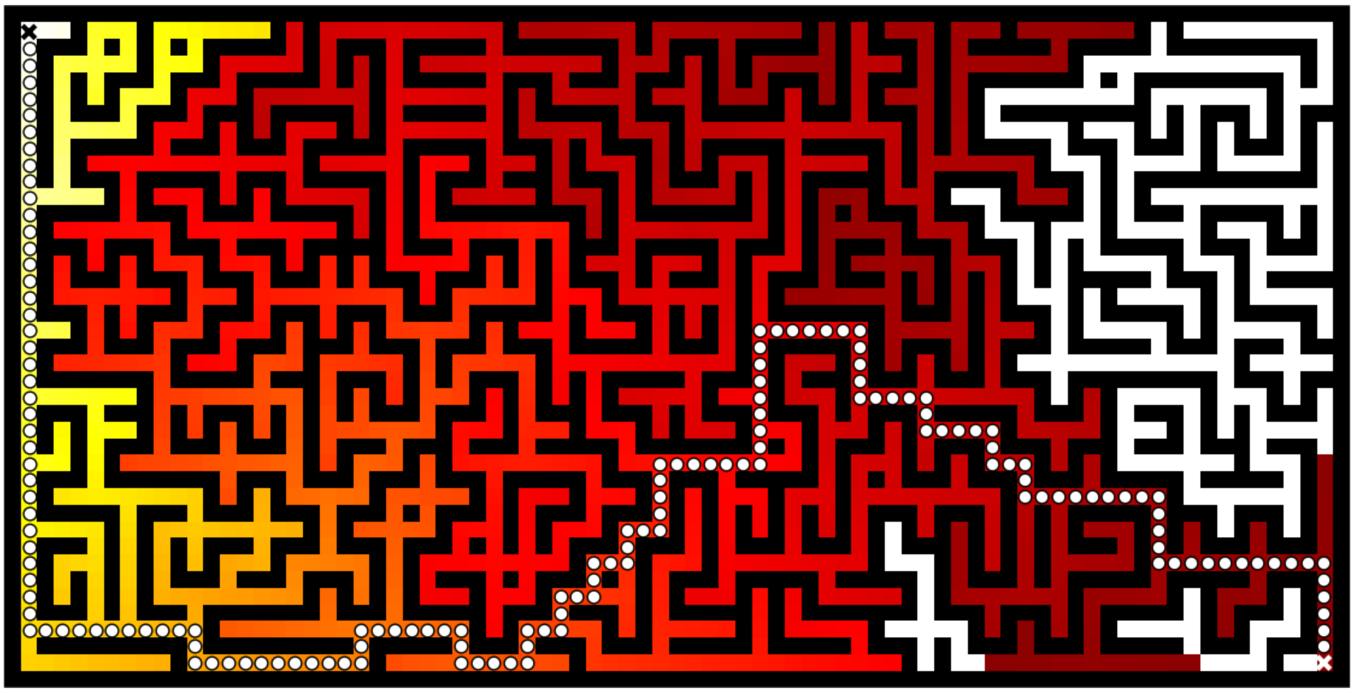
The NumPy implementation is straightforward if we take advantage of the `generic_filter` (from `scipy.ndimage`) for the diffusion process:

```
import numpy as np
def diffuse(Z):
  # North, West, Center, East, South
  return max(gamma*Z[0], gamma*Z[1], Z[2], gamma*Z[3], gamma*Z[4])

# Build gradient array
G = np.zeros(Z.shape)

# Initialize gradient at the entrance with value 1
G[start] = 1

# Discount factor
gamma = 0.99

# We iterate until value at exit is > 0. This requires the maze
# to have a solution or it will be stuck in the loop.
while G[goal] == 0.0:
  G = Z * generic_filter(G, diffuse, footprint=[[0, 1, 0],
                                                [1, 1, 1],
                                                [0, 1, 0]])
```

But in this specific case, it is rather slow. We'd better cook-up our own solution, reusing part of the game of life code:

```
import numpy as np

# Build gradient array
G = np.zeros(Z.shape)

# Initialize gradient at the entrance with value 1
G[start] = 1

# Discount factor
gamma = 0.99

# We iterate until value at exit is > 0. This requires the maze
# to have a solution or it will be stuck in the loop.
G_gamma = np.empty_like(G)
while G[goal] == 0.0:
    np.multiply(G, gamma, out=G_gamma)
    N = G_gamma[0:-2,1:-1]
    W = G_gamma[1:-1,0:-2]
    C = G[1:-1,1:-1]
    E = G_gamma[1:-1,2:]
    S = G_gamma[2:,1:-1]
    G[1:-1,1:-1] = Z[1:-1,1:-1]*np.maximum(N,np.maximum(W,
                                np.maximum(C,np.maximum(E,S))))
```

Once this is done, we can ascend the gradient to find the shortest path as illustrated on the figure below:



Path finding using the Bellman-Ford algorithm. Gradient colors indicate propagated values from the end-point of the maze (bottom-right). Path is found by ascending gradient from the goal.

Complete Solution

Let’s, once again, visualize the maze by replacing Breadth-First implementation with Bellman-Ford’s. 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 BellmanFord(Z, start, goal):


    # We reserve Z such that walls have value 0
    Z = 1 - Z

    # Build gradient array
    G = np.zeros(Z.shape)

    # Initialize gradient at the entrance with value 1
    G[start] = 1

    # Discount factor
    gamma = 0.99

    def diffuse(Z):
        # North, West, Center, East, South
        return max(gamma*Z[0], gamma*Z[1], Z[2], gamma*Z[3], gamma*Z[4])

    # Shortest path in best case cannot be less the Manhattan distance
    # from entrance to exit
    length = Z.shape[0]+Z.shape[1]

    # We iterate until value at exit is > 0. This requires the maze
    # to have a solution or it will be stuck in the loop.

    G_gamma = np.empty_like(G)
    while G[goal] == 0.0:
        # Slow
        # G = Z * generic_filter(G, diffuse, footprint=[[0, 1, 0],
        #                                               [1, 1, 1],
        #                                               [0, 1, 0]])

        # Fast
        np.multiply(G, gamma, out=G_gamma)
        N = G_gamma[0:-2,1:-1]
        W = G_gamma[1:-1,0:-2]
        C = G[1:-1,1:-1]
        E = G_gamma[1:-1,2:]
        S = G_gamma[2:,1:-1]
        G[1:-1,1:-1] = Z[1:-1,1:-1]*np.maximum(N,np.maximum(W,np.maximum(C,np.maximum(E,S))))

    # Descent gradient to find shortest path from entrance to exit
    y, x = goal
    P = []
    dirs = [(0,-1), (0,+1), (-1,0), (+1,0)]
    while (x, y) != start:
        P.append((x, y))
        neighbours = [-1, -1, -1, -1]
        if x > 0:
            neighbours[0] = G[y, x-1]
        if x < G.shape[1]-1:
            neighbours[1] = G[y, x+1]
        if y > 0:
            neighbours[2] = G[y-1, x]
        if y < G.shape[0]-1:
            neighbours[3] = G[y+1, x]
        a = np.argmax(neighbours)
        x, y  = x + dirs[a][1], y + dirs[a][0]
    P.append((x, y))
    return G, np.array(P)
```

```
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



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

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

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

    # 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.imshow(G, interpolation='nearest', cmap=cmap, vmin=0.01, vmax=G[start])
    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()
```

Further Readings

- Labyrinth Algorithms, Valentin Bryukhanov, 2014.