


Maze Finder Using OpenCV Python

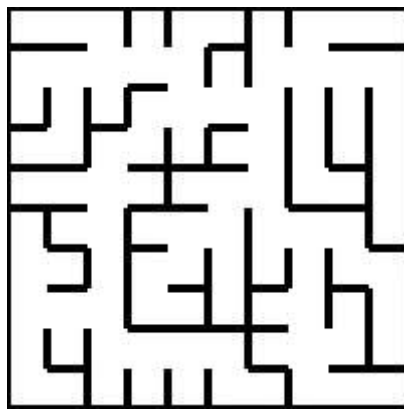


Om Rastogi · [Follow](#)

6 min read · Mar 28, 2020

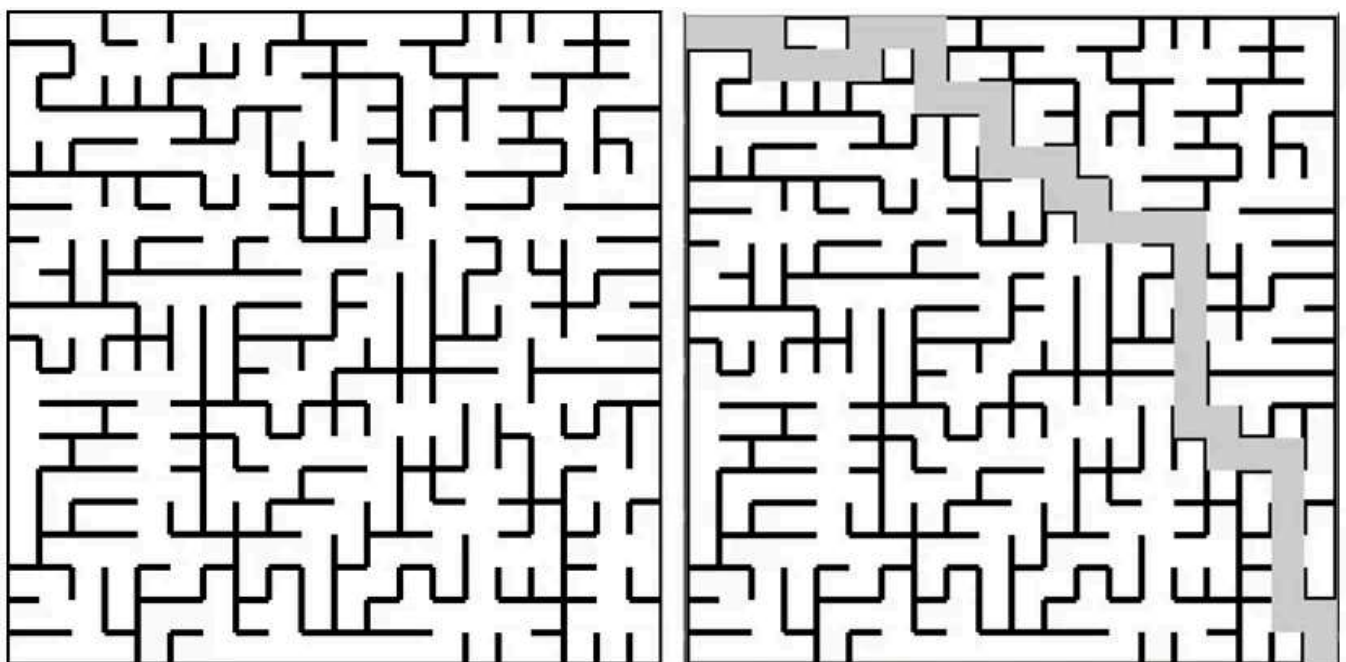
 Listen

 Share



An Easy maze problem

You must have seen maze puzzles in magazines and news paper clippings. Ever wondered, if your computer can do it for you. So today we are going to learn to make a maze puzzle solver using opencv in python.



First we import all the libraries-

```
import cv2
import numpy as np
import os
```

Let us start with reading the images:

```
def readImage(img_file_path):
    binary_img = None
    img = cv2.imread(img_file_path,0)
    ret,img = cv2.threshold(img,127,255,cv2.THRESH_BINARY)
    binary_img = img
    return binary_img
```

Now we have a problem of reading the maze block by block.



```
def blockwork(img,coordinate): size = CELL_SIZE
    h = CELL_SIZE*(coordinate[0]+1)
    w = CELL_SIZE*(coordinate[1]+1)
    h0= CELL_SIZE*coordinate[0]
    w0= CELL_SIZE*coordinate[1]

    block = img[h0:h,w0:w]

    up    = bool(block[0,int(size/2)]) *1000
    down  = bool(block[int(size-1),int(size/2)])*100
    left  = bool(block[int(size/2),0]) *10
    right = bool(block[int(size/2),int(size-1)])*1

    edge = up+down+left+right
    return edge, block
```

So the variable 'edge' contains the information of the the boundaries of every single block. We add value, if specific side of the block is open. This compresses the large information to just one number, $0 < \text{edge} < 1111$.



edge = 1



edge = 11



edge = 110

For upper edge — 1000 is added

For lower edge — 100 is added

For left edge — 10 is added

For right edge — 1 is added

Hence we can tell, which side of the given block is open and which side is closed.

```
def solveMaze(original_binary_img, initial_point, final_point,
no_cells_height, no_cells_width):
```

```
    edgearray = []
```

```
    for i in range (no_cells_height):
```

```
        edgearray.append([])
```

```
        for j in range(no_cells_width):
```

```
            sz = [i,j]
```

```
            edge, block = blockwork(img, sz)
```

```
            edgearray[i].append(edge)
```

```
    edge= edgearray
```

After getting the edge array of all the blocks, we append it in an edge array.

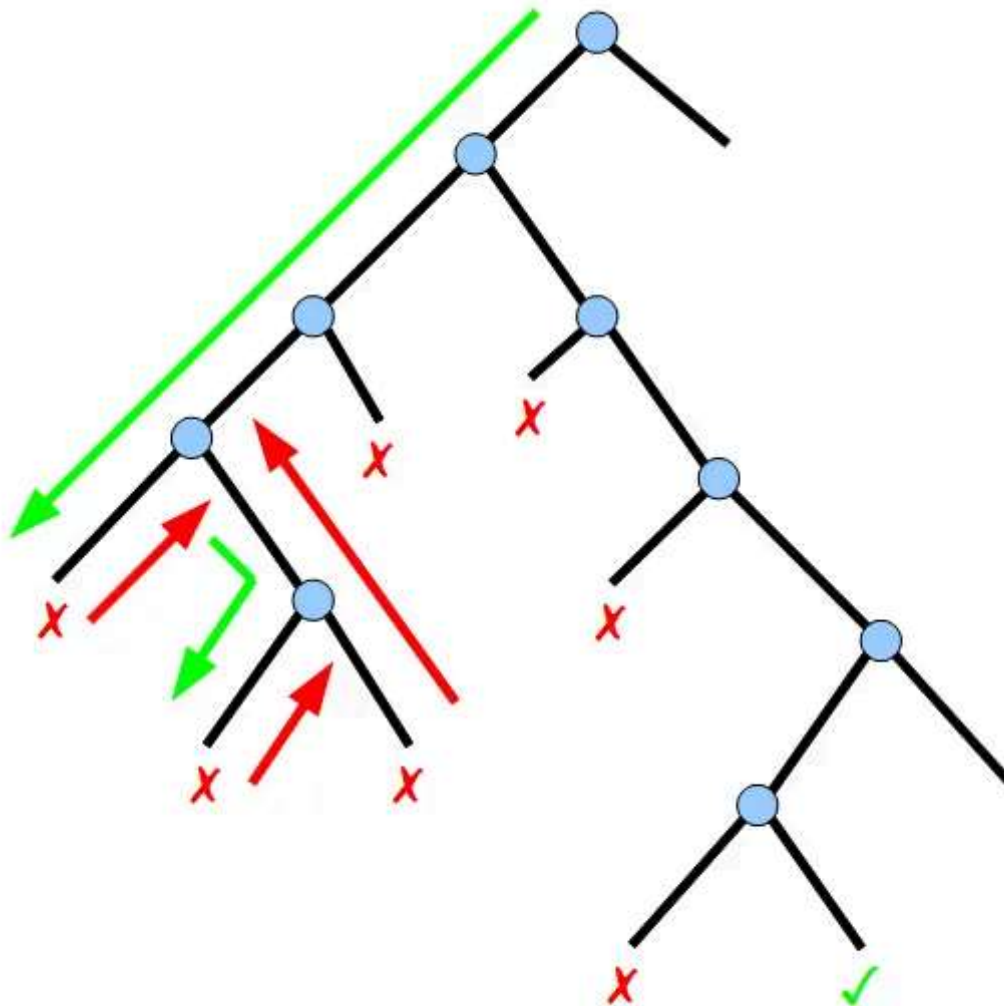
Solving the maze literally come down to solving this array —

```
[[1, 11, 110, 1, 110, 101, 111, 111, 10, 1, 11, 111, 11, 10, 100,
100, 101, 10, 1, 110], [101, 10, 1101, 111, 1111, 1010, 1000, 1100,
101, 10, 1, 1011, 110, 1, 1011, 1111, 1011, 10, 1, 1110], [1100, 1,
```

```
1010, 1000, 1000, 1, 111, 1011, 1011, 110, 101, 10, 1101, 10, 101,
1011, 10, 101, 11, 1010], [1101, 111, 11, 11, 111, 10, 1000, 100,
101, 1110, 1101, 10, 1100, 100, 1101, 10, 1, 1110, 1, 110], [1000,
1000, 1, 11, 1011, 11, 111, 1110, 1000, 1001, 1111, 110, 1001, 1111,
1011, 11, 10, 1100, 100, 1100], [1, 11, 110, 100, 1, 110, 1000,
1101, 10, 101, 1110, 1001, 110, 1101, 11, 10, 101, 1011, 1011,
1010], [1, 111, 1111, 1111, 11, 1011, 111, 1011, 110, 1000, 1000,
100, 1101, 1111, 11, 111, 1111, 111, 11, 10], [101, 1010, 1100,
1000, 1, 11, 1010, 1, 1011, 111, 11, 1111, 1110, 1101, 10, 1100,
1000, 1101, 11, 10], [1001, 10, 1000, 1, 111, 111, 111, 11, 11,
1110, 1, 1110, 1100, 1100, 101, 1111, 10, 1000, 101, 10], [1, 111,
111, 10, 1000, 1100, 1100, 1, 11, 1010, 101, 1110, 1100, 1000, 1000,
1101, 110, 101, 1011, 10], [100, 1000, 1100, 100, 100, 1100, 1100,
1, 110, 101, 1010, 1000, 1000, 1, 11, 1110, 1001, 1011, 11, 10],
[1101, 11, 1011, 1011, 1111, 1010, 1000, 1, 1111, 1010, 1, 110, 100,
101, 111, 1110, 101, 111, 11, 10], [1101, 11, 10, 1, 1111, 10, 1,
110, 1000, 100, 100, 1101, 1111, 1110, 1100, 1001, 1110, 1000, 100,
100], [1101, 10, 1, 11, 1111, 10, 100, 1001, 111, 1011, 1011, 1110,
1000, 1000, 1100, 100, 1101, 11, 1110, 1100], [1100, 101, 11, 11,
1111, 110, 1101, 111, 1010, 1, 111, 1110, 100, 1, 1111, 1010, 1000,
101, 1111, 1010], [1100, 1000, 1, 11, 1110, 1000, 1000, 1000, 1,
111, 1110, 1000, 1001, 111, 1111, 10, 100, 1000, 1100, 100], [1000,
1, 111, 111, 1011, 111, 10, 101, 11, 1110, 1000, 101, 11, 1110,
1101, 111, 1110, 1, 1111, 1110], [1, 110, 1100, 1100, 1, 1010, 100,
1000, 100, 1100, 101, 1010, 100, 1000, 1100, 1000, 1000, 1, 1110,
1000], [1, 1011, 1110, 1000, 101, 11, 1110, 1, 1110, 1000, 1000, 1,
1011, 111, 1111, 111, 10, 101, 1111, 110], [1, 11, 1011, 10, 1000,
1, 1011, 11, 1011, 11, 11, 11, 11, 1010, 1000, 1001, 10, 1000, 1000,
1000]]
```

Backtracking Algorithms

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time.



Working of Backtracking Algorithm

#The solvemaze method is continued here...

```

shortestPath = []
img = original_binary_img
sp = []
rec = [0]
p = 0
sp.append(list(initial_point))

while True:
    h,w = sp[p][0],sp[p][1]
#h stands for height and w stands for width
    if sp[-1]==list(final_point):
        break

    if edge[h][w] > 0:
        rec.append(len(sp))

    if edge[h][w]>999:
#If this edge is open upwards
        edge[h][w] = edge[h][w]-1000
        h = h-1
        sp.append([h,w])

```

```

    edge[h][w] = edge[h][w] - 100
    p = p + 1
    continue

    if edge[h][w] > 99:
#If the edge is open downward
        edge[h][w] = edge[h][w] - 100
        h = h + 1
        sp.append([h, w])
        edge[h][w] = edge[h][w] - 1000
        p = p + 1
        continue

    if edge[h][w] > 9:
#If the edge is open left
        edge[h][w] = edge[h][w] - 10
        w = w - 1
        sp.append([h, w])
        edge[h][w] = edge[h][w] - 1
        p = p + 1
        continue

    if edge[h][w] == 1:
#If the edge is open right
        edge[h][w] = edge[h][w] - 1
        w = w + 1
        sp.append([h, w])
        edge[h][w] = edge[h][w] - 10
        p = p + 1
        continue

    else:
#Removing the coordinates that are closed or don't show any path
        sp.pop()
        rec.pop()
        p = rec[-1]

    for i in sp:
        shortestPath.append(tuple(i))

return shortestPath

```

In the aforementioned code, we are iterating the edgarray using variable h,w. For each edge we are checking for the opening using the values.

We check each and every block that are connected to path or initial block. After the path is taken once, that path is deleted by subtracting respective value.

For backtracking, we have the variable 'rec'. For any block with multiple open path, it becomes a node/junction as it may lead to multiple paths. If the path is closed ahead say after some other blocks, than the initial node is accessed again. Since all

initially taken path, has been deleted for the node, new paths are accessed and checked for the completion of path.

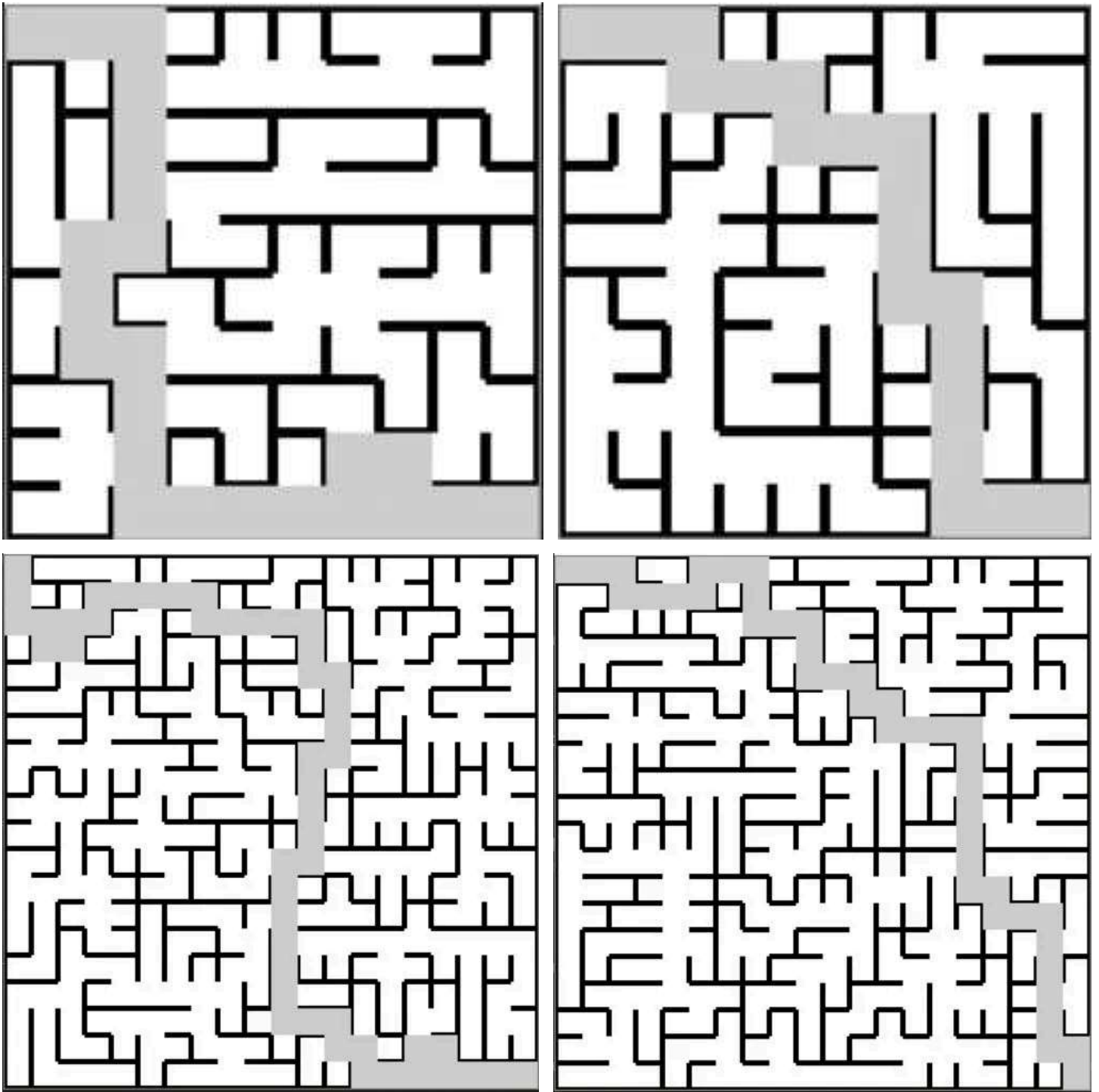
The shortest path is calculated as an array of all the point.

```
Shortest Path = [(0, 0), (0, 1), (0, 2), (1, 2), (1, 3), (1, 4), (1, 5), (0, 5), (0, 6), (0, 7), (1, 7), (2, 7), (2, 8), (2, 9), (3, 9), (4, 9), (4, 10), (4, 11), (5, 11), (5, 12), (6, 12), (6, 13), (6, 14), (6, 15), (7, 15), (8, 15), (9, 15), (10, 15), (11, 15), (12, 15), (12, 16), (13, 16), (13, 17), (13, 18), (14, 18), (15, 18), (16, 18), (17, 18), (18, 18), (18, 19), (19, 19)]
```

```
Length of Path = 41
```

To create the path on the image we have method **pathHighlight**. If you've understood the code above this is basically copied.

```
def pathHighlight(img, ip, fp, path):  
    size = CELL_SIZE  
    for coordinate in path:  
        h = CELL_SIZE*(coordinate[0]+1)  
        w = CELL_SIZE*(coordinate[1]+1)  
        h0= CELL_SIZE*coordinate[0]  
        w0= CELL_SIZE*coordinate[1]  
        img[h0:h,w0:w] = img[h0:h,w0:w]-50  
    return img
```



Hope, you enjoy this fun project and may find help in creating some sophisticated project.

Computer Vision

Python

Opencv

Maze

AI