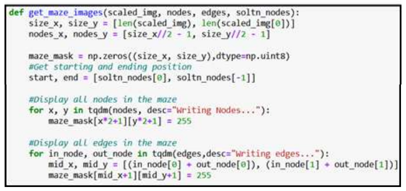
Highly Customizable Maze Generator Using Computer Vision

Developed and Presented by Mason Godfrey

GitHub Link: https://github.com/OverworldLord/computer-vision-maze-generator

Abstract

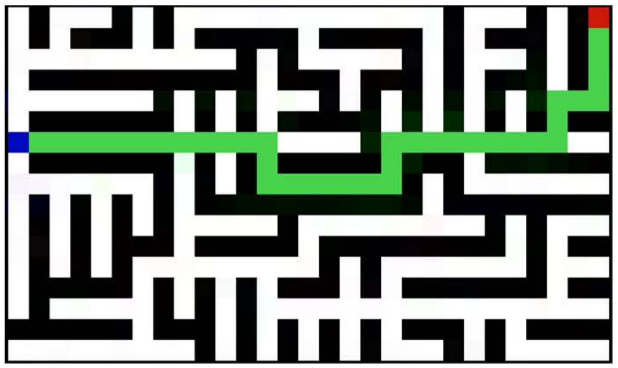
While computer vision is sometimes used when generating mazes, it is often very limited. Many websites, for example, allow for an input image to generate the external outline of a maze. This program looks to generate mazes that rely heavily on the input image by both using the input image to determine the shape of the maze and by generating the path of the maze based on the input image. Since generating the maze based on the input image can be slow for extremely large mazes, the program also has optional functionality that allows for sampling nodes which drastically improves the performance of the algorithm. The results of this program are three different output images. One output image contains the maze only, another adds a solution path to the maze, and the third image underlays the maze with a scaled version of the input image.



```
def get_maze_images(scaled_img, nodes, edges, soltn_nodes):
    size_x, size_y = [len(scaled_img), len(scaled_img[0])]
    nodes_x, nodes_y = [size_x//2 - 1, size_y//2 - 1]

    maze_mask = np.zeros((size_x, size_y),dtype=np.uint8)
    #Get starting and ending position
    start, end = [soltn_nodes[0], soltn_nodes[-1]]

    #Display all nodes in the maze
    for x, y in tqdm(nodes, desc="Writing Nodes..."):
        maze_mask[x*2+1][y*2+1] = 255

    #Display all edges in the maze
    for in_node, out_node in tqdm(edges,desc="Writing edges..."):
        mid_x, mid_y = [(in_node[0] + out_node[0]), (in_node[1] + out_node[1])]
        maze_mask[mid_x+1][mid_y+1] = 255
```

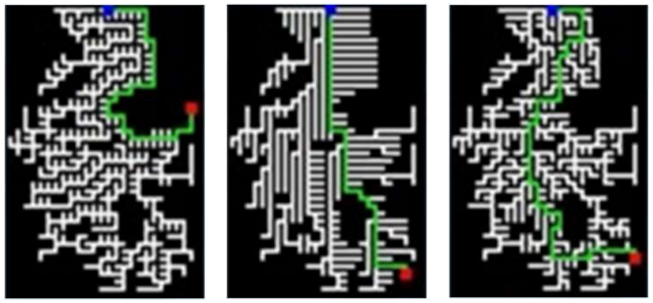
Maze Generation as a Graph Problem

Graphs consist of both nodes and edges. In normal graphs, nodes can connect to anywhere between zero and infinite nodes using edges, while edges connect exactly two nodes. Mazes are often a subset of the graph problem, meaning some restrictions on nodes and edges exist. In the case of this program, the restrictions are that each node can only connect to between one and four logically adjacent nodes and that all edges are bidirectional. Additionally, any location in the maze must be reachable from any other location in the maze. The physical version of the maze consists of nodes and edges and designates the maze's entrance and exit as nodes on the graph. The starting location is chosen randomly while the ending location is the last node to be added to the graph. Since every location of the maze is reachable from every other location of the maze, a solution path between the entrance and exit of the maze always exists (and is found immediately after the maze has been generated).



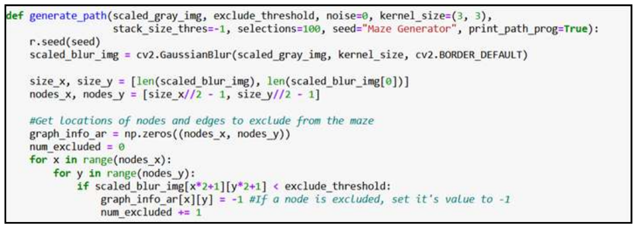
Algorithm Selection

In order to generate a maze where any location is reachable by any other location, the maze is generated by adding a single node to a graph and expanding outwards from that node until every node that can be directly or indirectly reached from the source node has been travelled to. While both the breadth first search (BFS) and depth first search (DFS) algorithms can be used to accomplish this, they result in uninteresting mazes that are extremely easy to solve. This is why the program's weighted expansion algorithm, which chooses a node to travel to randomly based on the weights associated with each node, is preferable to the other algorithms. Shown below from left to right are the BFS and DFS algorithms followed by this program's algorithm.

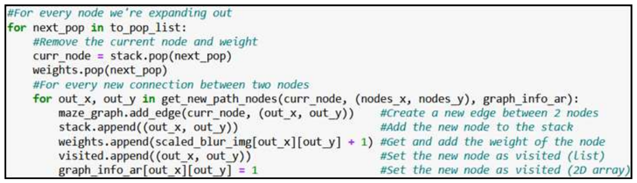


Computer Vision Paired With Maze Generation

The maze generation algorithm relies heavily on the input image for maze generation and image generation. The color values of the input image determines the weights used in path generation and whether the locations should be used in the maze (if the color value is lower than a threshold value). Other options allow for histogram equalization and different gaussian kernel sizes which affect the weight values that will be assigned to each cell. The below code demonstrates how these parameters affect maze generation. The two code cells are both from the 'generate_path' function, which handles the logical generation of the maze (while the 'get_maze_image' function turns the resulting maze into an output image).



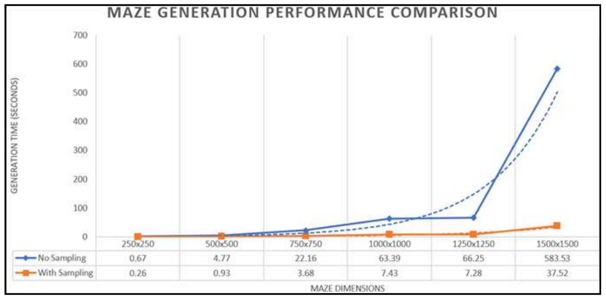
```
def generate_path(scaled_gray_img, exclude_threshold, noise=0, kernel_size=(3, 3),
                  stack_size_thres=-1, selections=100, seed="Maze Generator", print_path_prog=True):
    r.seed(seed)
    scaled_blur_img = cv2.GaussianBlur(scaled_gray_img, kernel_size, cv2.BORDER_DEFAULT)

    size_x, size_y = [len(scaled_blur_img), len(scaled_blur_img[0])]
    nodes_x, nodes_y = [size_x//2 - 1, size_y//2 - 1]

    #Get locations of nodes and edges to exclude from the maze
    graph_info_ar = np.zeros((nodes_x, nodes_y))
    num_excluded = 0
    for x in range(nodes_x):
        for y in range(nodes_y):
            if scaled_blur_img[x*2+1][y*2+1] < exclude_threshold:
                graph_info_ar[x][y] = -1 #If a node is excluded, set it's value to -1
                num_excluded += 1
```



```
#For every node we're expanding out
for next_pop in to_pop_list:
    #Remove the current node and weight
    curr_node = stack.pop(next_pop)
    weights.pop(next_pop)
    #For every new connection between two nodes
    for out_x, out_y in get_new_path_nodes(curr_node, (nodes_x, nodes_y), graph_info_ar):
        maze_graph.add_edge(curr_node, (out_x, out_y))    #Create a new edge between 2 nodes
        stack.append((out_x, out_y))                      #Add the new node to the stack
        weights.append(scaled_blur_img[out_x][out_y] + 1) #Get and add the weight of the node
        visited.append((out_x, out_y))                    #Set the new node as visited (list)
        graph_info_ar[out_x][out_y] = 1                   #Set the new node as visited (2D array)
```

Resources vs Generation Speed

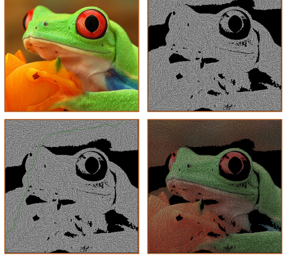
While the generator always works very quickly during the generation of small mazes, the larger the requested output maze the longer generation will take due to the time it takes to take make a weighted random selection. While this maze generator can use this method to create mazes of any size, the generator can optionally sample multiple locations near the selected weight in order to drastically reduce the speed at which mazes can be generated. Sampling only occurs when the number of possible choices for weights is above a 'stack size threshold' value which can be chosen by the user. The below chart shows how long it takes the program to generate mazes of different sizes both with and without sampling using a stack size threshold of 250 and sampling 100 of the values in the stack.

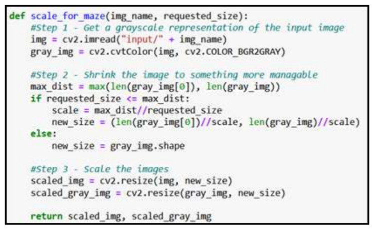


	250x250	500x500	750x750	1000x1000	1250x1250	1500x1500
No Sampling	0.67	4.77	22.16	63.39	66.25	583.53
With Sampling	0.26	0.93	3.68	7.43	7.28	37.52

Output

The below images demonstrate that the input image is used to generate three output images; the maze with the starting and ending points, the maze with the shortest possible solution path from the start to the end, and the maze overlayed onto a scaled version of the input image. The 'scale_for_maze' function takes the original input image and scales it to a user-requested size. In this case, the requested size is very large to demonstrate how thresholding affects the output image.





```
def scale_for_maze(img_name, requested_size):
    #Step 1 - Get a grayscale representation of the input image
    img = cv2.imread("input/" + img_name)
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    #Step 2 - Shrink the image to something more managable
    max_dist = max(len(gray_img[0]), len(gray_img))
    if requested_size <= max_dist:
        scale = max_dist//requested_size
        new_size = (len(gray_img[0])//scale, len(gray_img)//scale)
    else:
        new_size = gray_img.shape

    #Step 3 - Scale the images
    scaled_img = cv2.resize(img, new_size)
    scaled_gray_img = cv2.resize(gray_img, new_size)

    return scaled_img, scaled_gray_img
```

Conclusion

When used during maze generation, computer vision allows for the creation of unique and intricate mazes based on an initial input image. While this is not the first time that computer vision has been used for maze generation, the computer vision component of alternative maze generators usually only make use of the outline of an image. This approach allows users a wide variety of customization options to ensure that the generated mazes are both visually appealing and can be shaped to fit the user's preferences in a notably short amount of time even if the requested maze size is extremely large. As such, even the mazes generated to be too large for humans to solve are often aesthetically pleasing.