MAZE SOLVER

A PROJECT

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE AWARD OF THE DEGREE

OF

MASTER OF TECHNOLOGY

IN

INFORMATION SYSTEMS

Submitted by:

HIMANSHU VERMA

2K21/ISY/09

YASHPAL SINGH

2K21/ISY/25

Under the supervision of

Dr. JASRAJ MEENA

DEPARTMENT OF INFORMATION TECHNOLOGY

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi -110042

CHAPTER 1

INTRODUCTION

As we know that a **Maze** is a path or collection of paths, typically from an entrance to a goal. Mazes are often simple puzzles for humans, but they present a great programming problem that we can solve using shortest-path techniques like **Dijkstra's algorithm**.

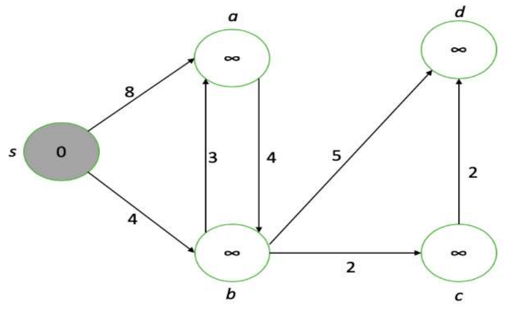
The way we are implementing it in our project is first we can use OpenCV, a popular computer vision library for Python, to extract pixel values and show our maze images, then we create a matrix of Vertices, representing the 2D layout of pixels in an image. This will be the basis for Dijkstra's algorithm graph. And second, we maintain a min-heap priority queue to keep track of unprocessed nodes. And after that, we start implementing Dijkstra's algorithm in such a way that it can work for all the mazes that we are going to capture using OpenCV.

1.1. DIJKSTRA'S ALGORITHM

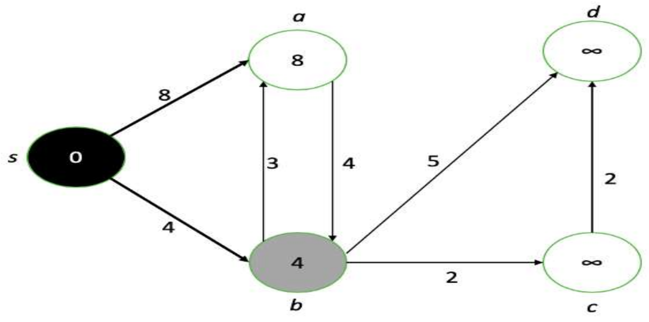
Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.

ALGORITHM:

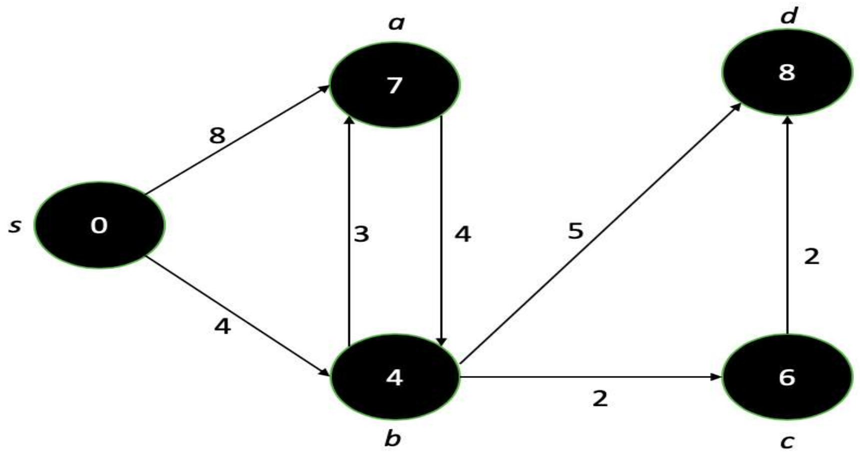
Step 1: We first assign a distance-from-source value to all the nodes. Node s receives a 0 value because it is the source; the rest receive values of ∞ to start. And also create a set that keeps tracks of the vertices included in the shortest path of the tree.



Step 2: First, we "relax" each adjacent vertex to our node of interest, updating their values to the minimum of their current value or the node of interest's value plus the connecting edge length**.**

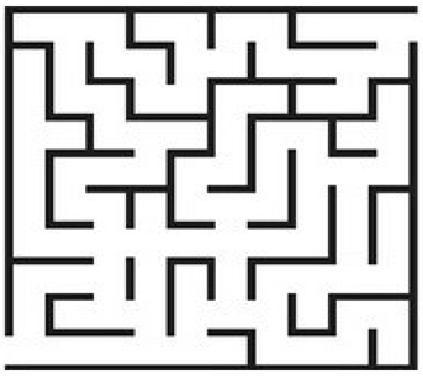


Step 3: Repeat the step 2 for all the vertices. And after going through each node, we eventually end up with a graph showing the shortest path length from the source to every node.



This is our final graph after running Dijkstra's algorithm for all the vertices.

1.2. BASIC IDEA OF THE PROJECT



We can think of an image as a matrix of pixels. Each pixel (for simplicity's sake) has an RGB value of 0,0,0 (black) or 255,255,255 (white). Our goal is to create a shortest path which starts in the white and does not cross into the black boundaries. To represent this goal, we can treat each pixel as a node and draw edges between neighboring pixels with edge lengths based on RGB value differences. We will use the Euclidean squared distance formula and add 0.1 to ensure no 0-distance path lengths (a requirement for Dijkstra's algorithm):

$$distance = 0.1 + (R_2 - R_1)^2 + (G_2 - G_1)^2 + (B_2 - B_1)^2$$

This formula makes the distance of crossing through the maze boundary prohibitively large. As we can see, the shortest path from source to destination will clearly be around the barrier, not through it.

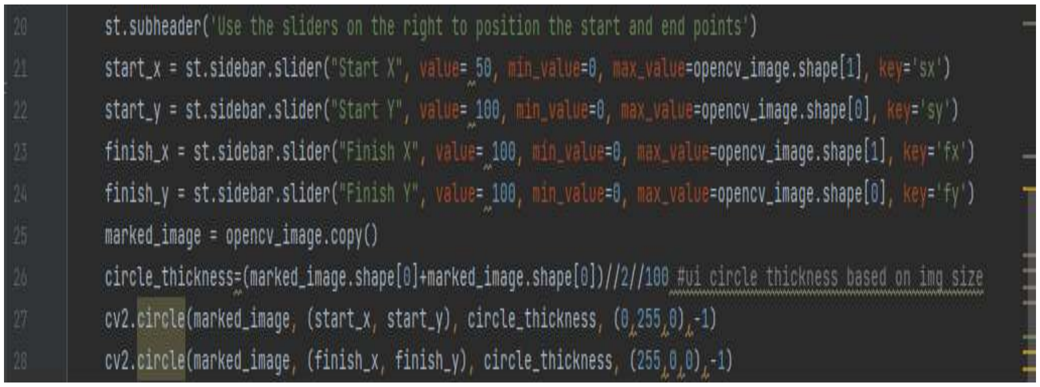
CHAPTER 2

PROJECT IMPLIMENTATION

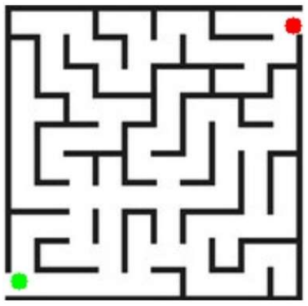
2.1. HELPER FUNCTIONS

We can use OpenCV, a popular computer vision library for Python, to extract pixel values and show our maze images. We identify the coordinates of our starting and ending locations by adding points to the maze, and user can shift these points according to the maze starting and the ending location with help of the side bar.

With the help of side bar user is able locate the starting and ending points of the maze by fixing the positions of x & y coordinate.

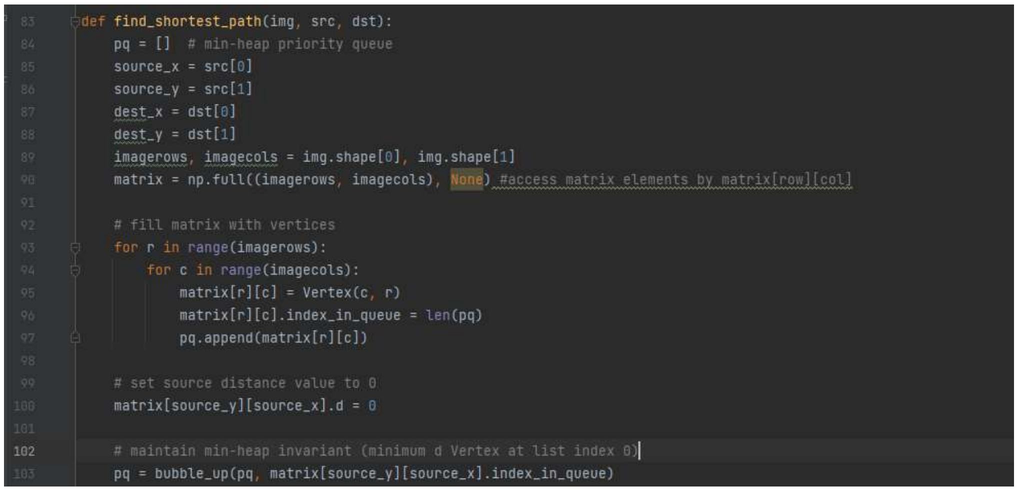


```
st.subheader('Use the sliders on the right to position the start and end points')
start_x = st.sidebar.slider("Start X", value= 50, min_value=0, max_value=opencv_image.shape[1], key='sx')
start_y = st.sidebar.slider("Start Y", value= 100, min_value=0, max_value=opencv_image.shape[0], key='sy')
finish_x = st.sidebar.slider("Finish X", value= 100, min_value=0, max_value=opencv_image.shape[1], key='fx')
finish_y = st.sidebar.slider("Finish Y", value= 100, min_value=0, max_value=opencv_image.shape[0], key='fy')
marked_image = opencv_image.copy()
circle_thickness=(marked_image.shape[0]+marked_image.shape[0])//2//100 #ui circle thickness based on img size
cv2.circle(marked_image, (start_x, start_y), circle_thickness, (0,255,0),-1)
cv2.circle(marked_image, (finish_x, finish_y), circle_thickness, (255,0,0),-1)
```



We have created a Vertex class which will help us keep track of the coordinates, because we also want to keep track of the parent node so that we can reconstruct the entire path once we find our distance values.

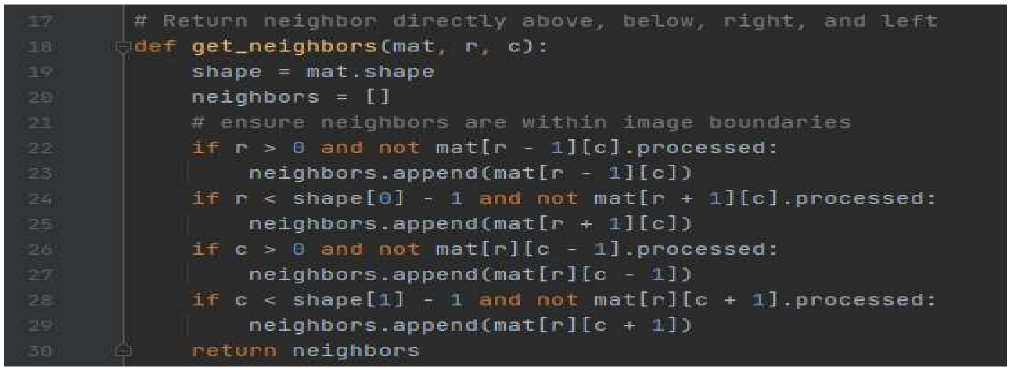
We also need to create a matrix of Vertices, representing the 2D layout of pixels in an image. This will be the basis for our Dijkstra's algorithm graph. We also maintain a min-heap priority queue to keep track of unprocessed nodes :



```python
def find_shortest_path(img, src, dst):
    pq = []  # min-heap priority queue
    source_x = src[0]
    source_y = src[1]
    dest_x = dst[0]
    dest_y = dst[1]
    imagerows, imagecols = img.shape[0], img.shape[1]
    matrix = np.full((imagerows, imagecols), None) #access matrix elements by matrix[row][col]

    # fill matrix with vertices
    for r in range(imagerows):
        for c in range(imagecols):
            matrix[r][c] = Vertex(c, r)
            matrix[r][c].index_in_queue = len(pq)
            pq.append(matrix[r][c])

    # set source distance value to 0
    matrix[source_y][source_x].d = 0

    # maintain min-heap invariant (minimum d Vertex at list index 0)
    pq = bubble_up(pq, matrix[source_y][source_x].index_in_queue)
```

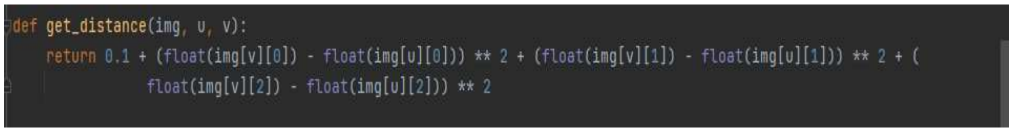
Now, after that we need some helper functions to find edges and edge lengths between vertices:

This function returns all the neighbors of the processing node:



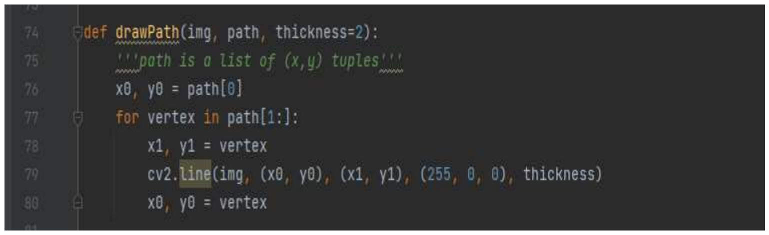
```python
# Return neighbor directly above, below, right, and left
def get_neighbors(mat, r, c):
    shape = mat.shape
    neighbors = []
    # ensure neighbors are within image boundaries
    if r > 0 and not mat[r - 1][c].processed:
        neighbors.append(mat[r - 1][c])
    if r < shape[0] - 1 and not mat[r + 1][c].processed:
        neighbors.append(mat[r + 1][c])
    if c > 0 and not mat[r][c - 1].processed:
        neighbors.append(mat[r][c - 1])
    if c < shape[1] - 1 and not mat[r][c + 1].processed:
        neighbors.append(mat[r][c + 1])
    return neighbors
```

This is the implementation of the Euclidean squared distance formula:



```python
def get_distance(img, u, v):
    return 0.1 + (float(img[v][0]) - float(img[u][0])) ** 2 + (float(img[v][1]) - float(img[u][1])) ** 2 + (
            float(img[v][2]) - float(img[u][2])) ** 2
```

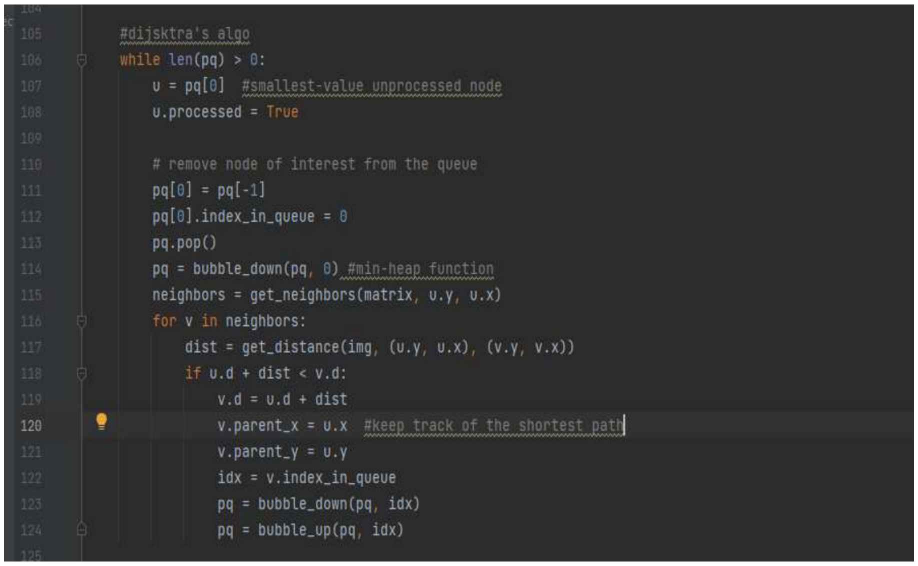
And at the last we have a drawpath function which is used to draw the whole path of the maze from starting to finishing position:



```python
def drawPath(img, path, thickness=2):
    '''path is a list of (x,y) tuples'''
    x0, y0 = path[0]
    for vertex in path[1:]:
        x1, y1 = vertex
        cv2.line(img, (x0, y0), (x1, y1), (255, 0, 0), thickness)
        x0, y0 = vertex
```

2.2. DIJKSTRA'S ALGORITHM IMPLEMENTATION

This is the implementation of Dijkstra's algorithm in our project code:

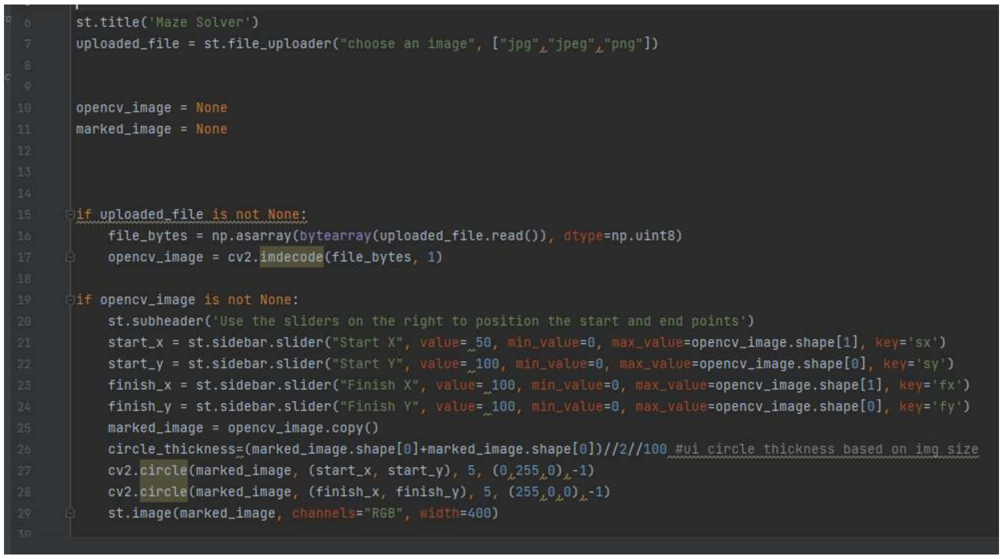


```python
#dijsktra's algo
while len(pq) > 0:
    u = pq[0]  #smallest-value unprocessed node
    u.processed = True

    # remove node of interest from the queue
    pq[0] = pq[-1]
    pq[0].index_in_queue = 0
    pq.pop()
    pq = bubble_down(pq, 0) #min-heap function
    neighbors = get_neighbors(matrix, u.y, u.x)
    for v in neighbors:
        dist = get_distance(img, (u.y, u.x), (v.y, v.x))
        if u.d + dist < v.d:
            v.d = u.d + dist
            v.parent_x = u.x  #keep track of the shortest path
            v.parent_y = u.y
            idx = v.index_in_queue
            pq = bubble_down(pq, idx)
            pq = bubble_up(pq, idx)
```

2.3. GUI IMPLEMENTATION FOR MAZE

With the help of **streamlit** library in python we have created a UI for the image uploader . We can add text outputs using functions like st.write() or st.title(). We store a dynamically uploaded file using streamlit's st.file_uploader() function. Finally, st.checkbox() will return a boolean based on whether the user has selected the checkbox.

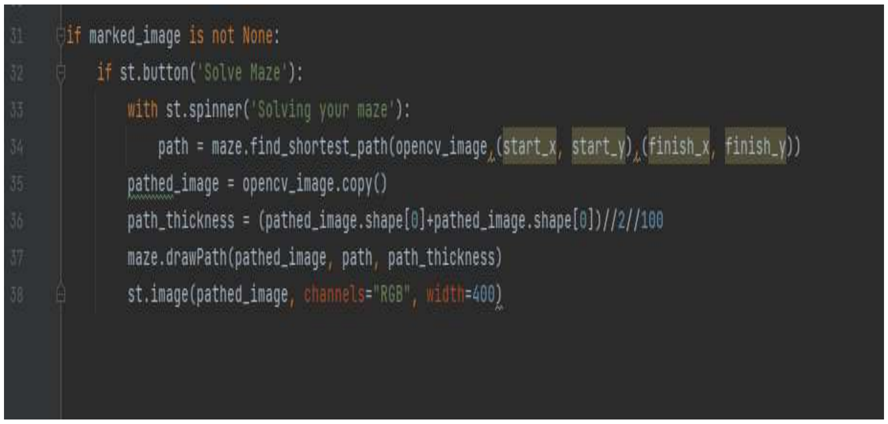
Once an image is uploaded, we want to show the image marked up with starting and ending points. We will use sliders to allow the user to reposition these points. The st.sidebar() function adds a sidebar to the page and st.slider() takes numerical input within a defined minimum and maximum. We can define slider minimum and maximum values dynamically based on the size of our maze image.



```python
st.title('Maze Solver')
uploaded_file = st.file_uploader("choose an image", ["jpg","jpeg","png"])


opencv_image = None
marked_image = None



if uploaded_file is not None:
    file_bytes = np.asarray(bytearray(uploaded_file.read()), dtype=np.uint8)
    opencv_image = cv2.imdecode(file_bytes, 1)

if opencv_image is not None:
    st.subheader('Use the sliders on the right to position the start and end points')
    start_x = st.sidebar.slider("Start X", value= 50, min_value=0, max_value=opencv_image.shape[1], key='sx')
    start_y = st.sidebar.slider("Start Y", value= 100, min_value=0, max_value=opencv_image.shape[0], key='sy')
    finish_x = st.sidebar.slider("Finish X", value= 100, min_value=0, max_value=opencv_image.shape[1], key='fx')
    finish_y = st.sidebar.slider("Finish Y", value= 100, min_value=0, max_value=opencv_image.shape[0], key='fy')
    marked_image = opencv_image.copy()
    circle_thickness=(marked_image.shape[0]+marked_image.shape[0])//2//100 #ui circle thickness based on img size
    cv2.circle(marked_image, (start_x, start_y), 5, (0,255,0),-1)
    cv2.circle(marked_image, (finish_x, finish_y), 5, (255,0,0),-1)
    st.image(marked_image, channels="RGB", width=400)
```

Whenever the user adjusts the sliders, the image is quickly re-rendered and the points change position.

Once the user has finalized the start and end positions, we want a button to solve the maze and display the solution. The st.spinner() element is displayed only while its child process is running, and the st.image() call is used to display an image.

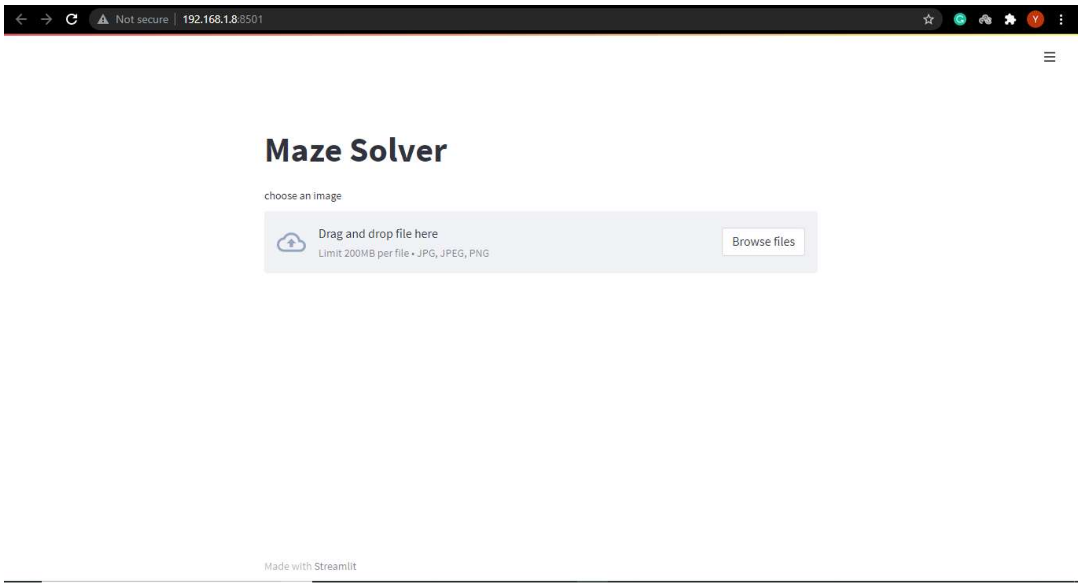


```python
if marked_image is not None:
    if st.button('Solve Maze'):
        with st.spinner('Solving your maze'):
            path = maze.find_shortest_path(opencv_image,(start_x, start_y),(finish_x, finish_y))
        pathed_image = opencv_image.copy()
        path_thickness = (pathed_image.shape[0]+pathed_image.shape[0])//2//100
        maze.drawPath(pathed_image, path, path_thickness)
        st.image(pathed_image, channels="RGB", width=400)
```

CHAPTER 3

EXPERIMENTAL RESULTS

3.1. THE MAIN SCREEN



Not secure | 192.168.1.8:8501

Maze Solver

choose an image

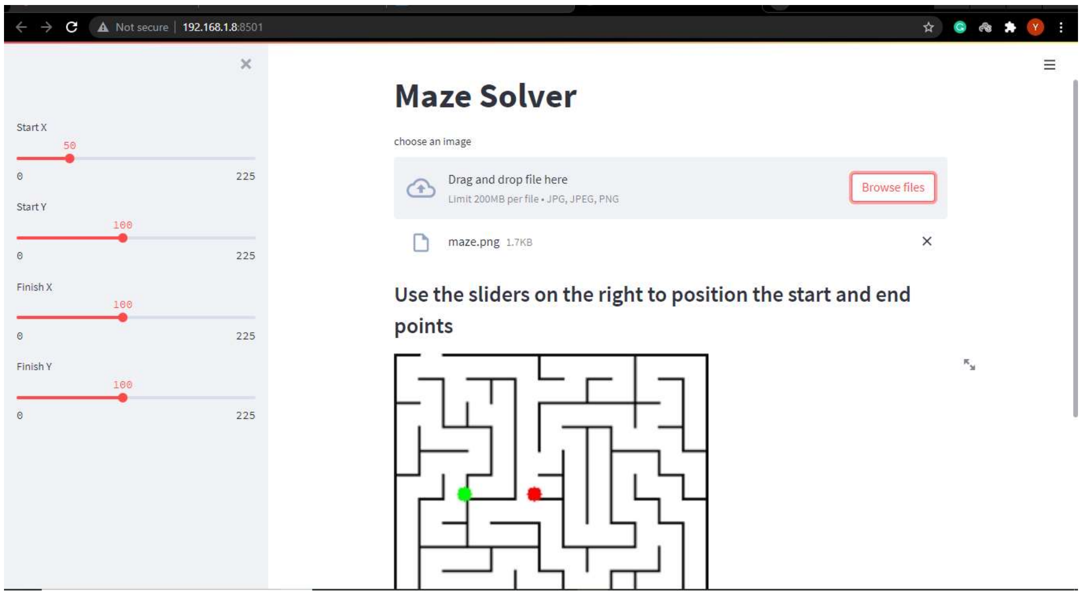
Drag and drop file here

Limit 200MB per file • JPG, JPEG, PNG

Browse files

Made with Streamlit

3.2. AFTER UPLOADING THE MAZE IMAGE



Not secure | 192.168.1.8:8501

Start X 50 0 225

Start Y 100 0 225

Finish X 100 0 225

Finish Y 100 0 225

Maze Solver

choose an image

Drag and drop file here

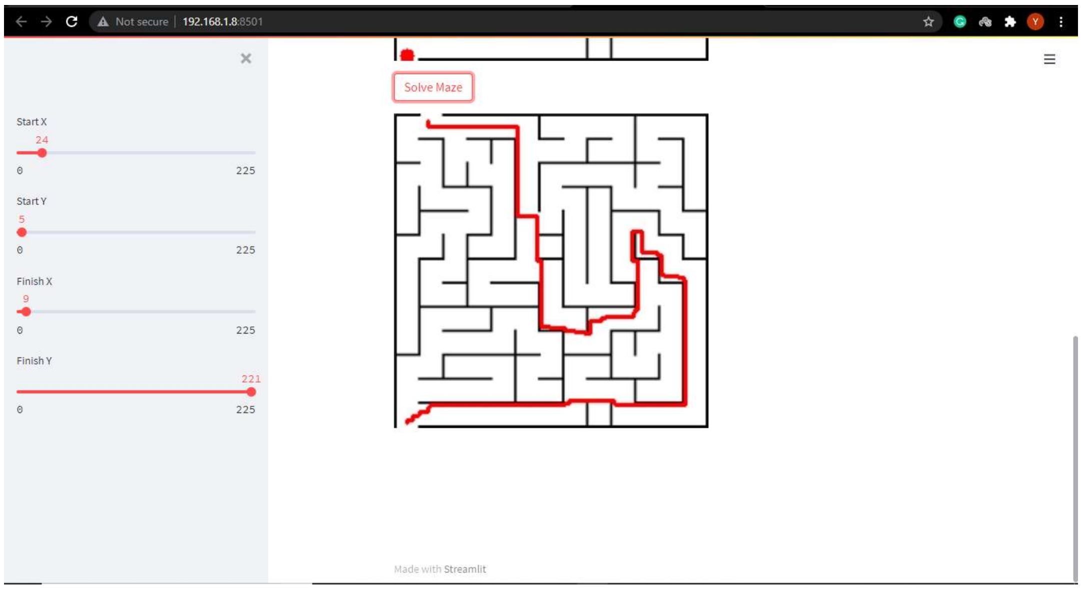
Limit 200MB per file • JPG, JPEG, PNG

Browse files

maze.png 1.7KB

Use the sliders on the right to position the start and end points

3.3. THE SOLVED MAZE



Start X 24 (0–225)

Start Y 5 (0–225)

Finish X 9 (0–225)

Finish Y 221 (0–225)

These are the output screens of our project.

CONCLUSION

We have created a maze solver by using Dijkstra’s algorithm for finding the shortest path. And also use openCV for processing the maze image and fetch out the pixel values of the white and black color, these values we used to detect the boundaries in the maze. After that we also implemented a simple UI for the maze solver. We did not need to write any traditional front-end code. Besides Streamlit’s ability to digest simple Python code, Streamlit intelligently re-runs the necessary parts of our script from top to bottom whenever the user interacts with the page or the script is changed. This allows for straightforward data flow and rapid development.

REFERENCES

[1] https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

[2] Murata, Yoshitaka, and Yoshihiro Mitani. "A study of shortest path algorithms in maze images." In SICE Annual Conference 2011, pp. 32-33. IEEE, 2011.