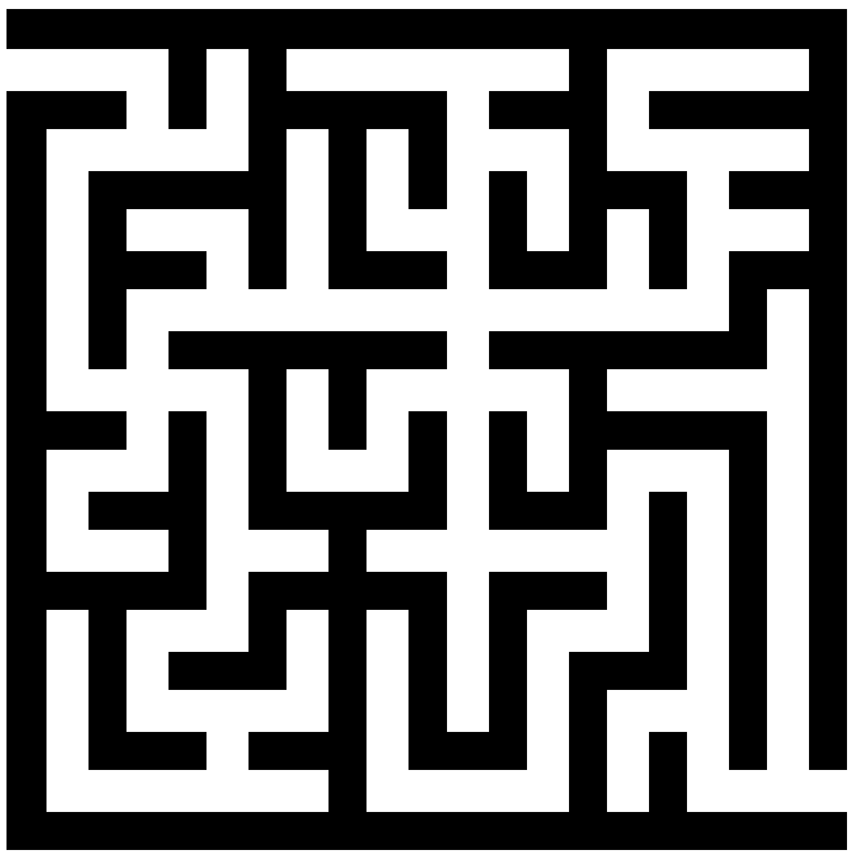
Path finding in a maze using A* algorithm

Explanation of the code (step by step procedure):

1. Opening the image:

```
from PIL import Image
img = Image.open(r"maze.png")
display(img)
```

Output:



2. Converting the image to grayscale – A common preprocessing step which simplifies the image while retaining the necessary information for maze-solving:

```
from IPython.display import Image, display
grayscale_img = img.convert("L")
```

```
grayscale_img.save("grayscale_img.png")
display(Image(filename="grayscale_img.png"))
```

3. Applying a threshold to convert the grayscale image into a binary image – A common technique to distinguish between walls and paths in a maze:

```
from IPython.display import Image, display
threshold_value = 128
binary_img = img.point(lambda p: p>threshold_value and
255)
binary_img.save("threshold_img.png")
display(Image(filename="threshold_img.png"))
```

4. Applying a median filter that removes small noise or artefacts from the image, further improving the quality of the thresholded image:

```
from PIL import Image, ImageFilter

img = Image.open("threshold_img.png")
filtered_img =
img.filter(ImageFilter.MedianFilter(size=3))
filtered_img.save("filtered_img.png")
from IPython.display import Image, display
display(Image(filename="filtered_img.png"))
```

5. Using OpenCV to find contours in the thresholded image – Drawing the contours on a copy of the image helps visualise the detected boundaries:

```
import cv2
import numpy as np
from IPython.display import Image, display

```

```
img =
cv2.imread("threshold_img.png",cv2.IMREAD_GRAYSCALE)
contours,_=cv2.findContours(img, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
contour_img = np.copy(img)
cv2.drawContours(contour_img, contours, -1, (255, 0,
0),1)
cv2.imwrite("contour_img.png",contour_img)
display(Image(filename="contour_img.png"))
```

6. Isolating the region of interest (ROI) by finding and filling the largest contour in the thresholded image – A crucial step in preparing the maze for the A* algorithm:

```
import cv2
import numpy as np
from IPython.display import Image, display

img =
cv2.imread("threshold_img.png",cv2.IMREAD_GRAYSCALE)
contours,_ = cv2.findContours(img, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
roi_img = np.copy(img)
def contour_area(c):
    return cv2.contourArea(c)
largest_contour = max(contours, key = contour_area)
cv2.drawContours(roi_img, [largest_contour],-1, (255,
255, 255),-1)
cv2.imwrite("roi_img.png",roi_img)
display(Image(filename="roi_img.png"))
```

7. Resizing the ROI image and implementing a function to map image coordinates to grid coordinates – This mapping is crucial for aligning the maze image with the internal representation used by the A* algorithm:

```python
import cv2
import numpy as np
roi_img = cv2.imread("roi_img.png",cv2.IMREAD_GRAYSCALE)
grid_width, grid_height = 20,20
resized_img = cv2.resize(roi_img,
(grid_width,grid_height))
def map_coordinates(image_x, image_y):
    grid_x = (image_x / roi_img.shape[1]) * grid_width
    grid_y = grid_height - (image_y / roi_img.shape[0]) *
grid_height
    return int(grid_x), int(grid_y)

image_x, image_y = 50, 30
grid_x, grid_y = map_coordinates(image_x, image_y)
print("Image coordinates : ", image_x,image_y)
print("Grid coordinates : ", grid_x, grid_y)
```

Output:

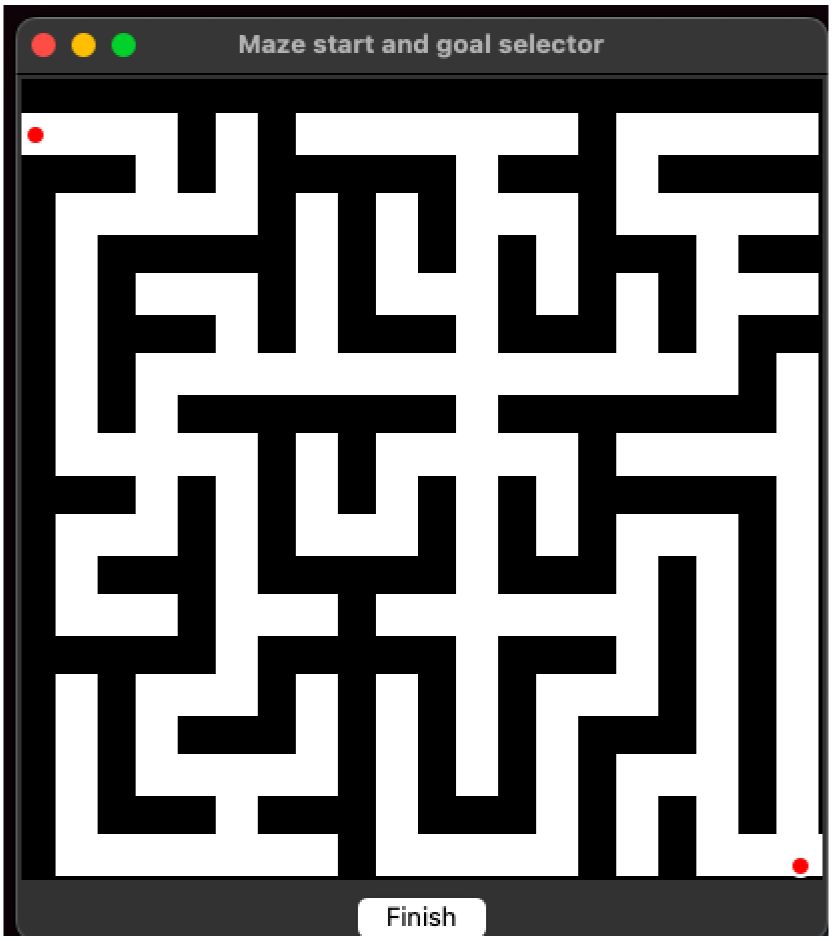
```
Image coordinates :  50 30
Grid coordinates :  2 18
```

8. Creating a simple interface for selecting the starting and goal coordinates on an image – Helps in defining the points for your maze-solving algorithm:

```python
import tkinter as tk
from tkinter import filedialog
from PIL import Image, ImageTk

window = tk.Tk()
window.title("Maze start and goal selector")

canvas = tk.Canvas(window, width=400, height=400)
canvas.pack()

```

```
file_path = filedialog.askopenfilename()
if file_path:
    image = Image.open(file_path)
    photo = ImageTk.PhotoImage(image)
    canvas.create_image(0, 0, anchor=tk.NW, image=photo)
else:
    print("No image selected.")

start_x, start_y = None, None
goal_x, goal_y = None, None

def on_canvas_click(event):
    global start_x, start_y, goal_x, goal_y
    x,y = event.x, event.y
    if start_x is None:
        start_x, start_y = x,y
        print("Start coordinates : ", start_x, start_y)
    elif goal_x is None:
        goal_x, goal_y = x,y
        print("Goal coordinates : ", goal_x, goal_y)

    canvas.create_oval(x-5,y-5,x+5,y+5, fill = "red",
width=2)
canvas.bind("<Button-1>",on_canvas_click)


def close_window():
    window.destroy()
    window.quit()

finish_button = tk.Button(window, text="Finish",
command=close_window)
finish_button.pack()

window.mainloop()
```

GUI interface:



(User clicks on the starting and end points, from which we get the starting and goal coordinates)

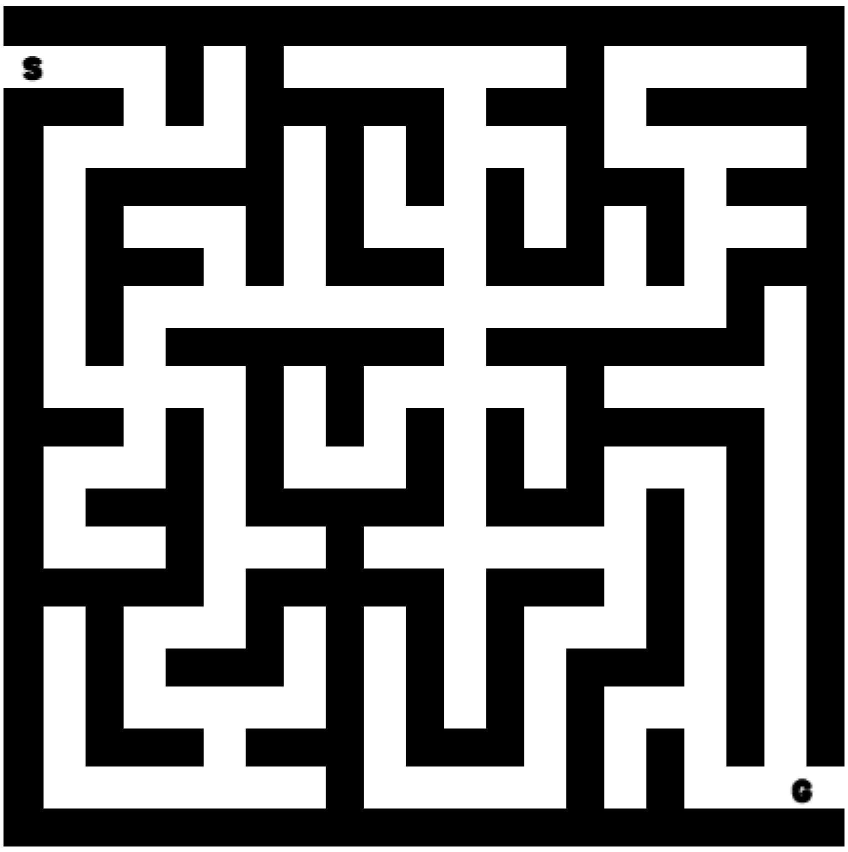
Output:

```
Start coordinates :  10 31
Goal coordinates :  392 396
```

9. Marking the start and goal locations on the resized image – The symbols 'S' and 'G' helps the algorithm identify the starting and goal points during the maze-solving process:

```
import cv2
import numpy as np
from IPython.display import Image, display

# Read the resized image
resized_image = cv2.imread("roi_img.png",
cv2.IMREAD_GRAYSCALE)  # Replace "resized_image.png" with
the path to your resized image

# Define the coordinates of the start and goal locations
start_coordinates = (start_x, start_y)
goal_coordinates = (goal_x, goal_y)  # Replace with the
grid coordinates of your goal location

# Define a function to mark locations with symbols
def mark_location(image, coordinates, text, color,
thickness):
    return cv2.putText(image, text, coordinates,
cv2.FONT_HERSHEY_SIMPLEX, 0.4, color, thickness,
cv2.LINE_AA)

# Create a copy of the resized image to mark start and
goal locations
marked_image = np.copy(resized_image)

# Set the text colours
start_color = (0, 255, 0)  # Green for start
goal_color = (0, 0, 255)  # Red for goal

# Mark the start and goal locations
marked_image = mark_location(marked_image,
start_coordinates, 'S', start_color, 2)
marked_image = mark_location(marked_image,
goal_coordinates, 'G', goal_color, 2)


# Save the marked image (optional)
cv2.imwrite("marked_image.png", marked_image)  # Replace
"marked_image.png" with the desired output file path
display(Image(filename="marked_image.png"))
```

Output:



(Displaying ‘S’ and ‘G’ at the starting and goal points)

10. The Python script uses the A* algorithm to find the shortest path through a maze represented as a 2D array. Steps involved:
    a. Image Processing
        i. Reads a grayscale image representing the maze.
        ii. Converts pixel values to maze cells ('X' for blocked, ' ' for open).
        iii. Marks the start ('S') and goal ('G') positions.
    b. Labelling
        i. Converts the grayscale image to a labelled image where 0 represents open spaces, 1 represents obstacles, 2 represents the start, and 3 represents the goal.
    c. A* Algorithm
        i. Defines a heuristic function for estimating the cost from a current position to the goal.
        ii. Uses a priority queue to explore positions based on their total cost (g_score + heuristic).

   iii. Updates the path and costs while exploring neighbours.
   iv. Stops when the goal is reached and reconstructs the path.

d. Path Finding
   i. Calls the A* algorithm with the marked maze array, start, and goal coordinates.
   ii. Prints the initial maze array and labelled image.
   iii. Outputs the found path from the start to the goal.

```
import heapq
import numpy as np
from skimage import io

# Load the image as grayscale
gray_image = io.imread("0.png", as_gray=True)

# Set threshold values
threshold_open = 0.85
threshold_blocked = 0.10

# Create the maze array
maze_array = [[' ' if pixel_value > threshold_open else
'X' if pixel_value < threshold_blocked else ' ' for
pixel_value in row] for row in gray_image]

# Set 'S' (start) and 'G' (goal) locations
start_coordinates = (start_x, start_y)  # Replace with
the actual coordinates of the start point
goal_coordinates = (goal_x, goal_y)   # Replace with the
actual coordinates of the goal point

# Create a new maze array with 'S' and 'G'
marked_maze_array = [row[:] for row in maze_array]
marked_maze_array[start_coordinates[1] //
20][start_coordinates[0] // 20] = 'S'
```

```
marked_maze_array[goal_coordinates[1] //
20][goal_coordinates[0] // 20] = 'G'

print("Initial Maze Array:")
for row in marked_maze_array:
    print(''.join(map(str, row)))

# Assume you have the labeled_image generated from the
segmentation
labeled_image = np.where(gray_image > threshold_open, 1,
0)  # Use 0 for open spaces, 1 for obstacles

# Mark 'S' and 'G' in the labeled image
labeled_image[start_coordinates[1] // 20,
start_coordinates[0] // 20] = 2  # Use 2 for the start
labeled_image[goal_coordinates[1] // 20,
goal_coordinates[0] // 20] = 3  # Use 3 for the goal

print("Labeled Image:")
print(labeled_image)

def astar_algorithm(maze, start, goal):
    def heuristic(current, goal):
        return np.sqrt((current[0] - goal[0]) ** 2 +
(current[1] - goal[1]) ** 2)


    open_list = [(0, start)]
    came_from = {}
   # Modify the initialization of g_score
    g_score = {(x, y): float('inf') for x in
range(len(maze[0])) for y in range(len(maze))}

    g_score[(start[0], start[1])] = 0

    while open_list:
        _, current = heapq.heappop(open_list)

        if current == goal:
            path = reconstruct_path(came_from, current)
            return path

        for neighbor in get_neighbors(current, maze):
            tentative_g_score = g_score[current] + 1  #
Assuming each step has a cost of 1

            if tentative_g_score < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score = tentative_g_score +
heuristic(neighbor, goal)
                heapq.heappush(open_list, (f_score,
neighbor))


def get_neighbors(current, maze):
    x, y = current
    neighbors = []

    # Define possible moves (up, down, left, right)
    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy

        # Check if the new position is within the maze
bounds
        if 0 <= new_x < len(maze[0]) and 0 <= new_y <
len(maze):
            # Check if the cell is open (you may need to
adapt this check based on your maze representation)
            if maze[new_y][new_x] != 'X':  # 'X'
```

```python
represents a blocked cell
                neighbors.append((new_x, new_y))

    return neighbors


def reconstruct_path(came_from, current):
    # Reconstruct the path from the 'came_from'
dictionary
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.insert(0, current)
    return path

# Find the path using the A* algorithm
path = astar_algorithm(marked_maze_array,
start_coordinates, goal_coordinates)

# Print the found path
print("Found Path:", path)
```

Output:

```
Initial Maze Array:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
SXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
XXXXXXXXXXXXXXXXXXX                     XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXX                     XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX

XXXXXXXXXXXXXXXXXXX                     XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXX
Labelled Image:
[[0 0 0 ... 0 0 0]
 [2 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 ...
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]]
Found Path: [(10, 35), (11, 35), (12, 35), (13, 35), (14, 35), (15, 35),
(16, 35), (17, 35), (18, 35), (19, 35), (20, 35), (21, 35), (22, 35), (23,
35), (24, 35), (25, 35), (26, 35), (27, 35), (28, 35), (29, 35), (30, 35),
(31, 35), (32, 35), (33,
35).........................................................................
.............(388, 391), (389, 391), (389, 392), (390, 392), (390, 393),
(391, 393), (391, 394), (392, 394), (392, 395), (393, 395), (393, 396),
(394, 396)]
```

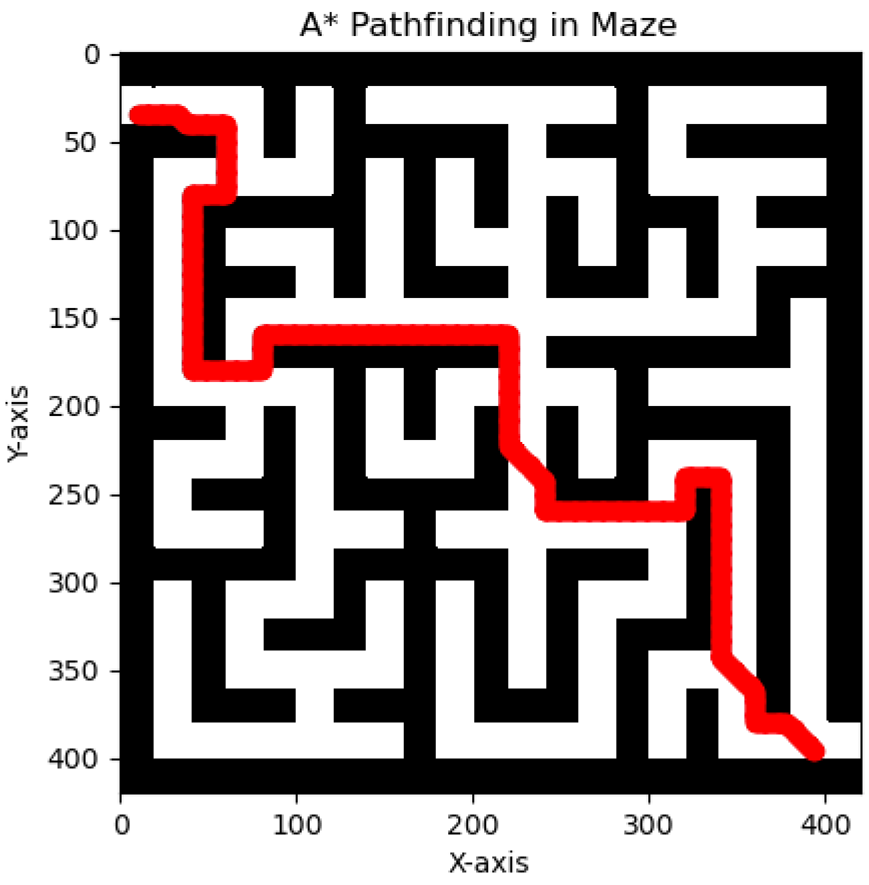
11. Using Matplotlib to visualise the maze along with the found path
    a. Colormap
        i. Defines a colormap (`cmap`) with colours for open spaces, obstacles, start, and goal.
    b. Maze Conversion
        i. Converts the maze array into a NumPy array with numerical values (0 for open, 1 for obstacles).
    c. Marking Start and Goal
        i. Marks the start and goal positions in the maze array.
    d. Plotting the Maze
        i. Uses `plt.imshow()` to plot the maze array with the specified colormap.
    e. Path Plotting
        i. Extracts x and y coordinates from the path and removes diagonal moves.

    ii. Plots the path as a red line with markers.

f. Labels and Title
    i. Sets labels for the axes and a title for the plot.
g. Saving and Displaying
    i. Saves the plot as an image ('maze_with_path.png').
    ii. Displays the plot.
h. The `remove_diagonal_moves` function ensures that diagonal moves are not plotted in the visualisation, giving a clearer representation of the path.

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import ListedColormap

def visualize_maze_with_path(maze, start, goal, path):
    # Swap black and white colours in the colormap
    cmap = ListedColormap(['white', 'black', 'green',
'red'])

    # Convert maze_array to a NumPy array with numerical
values
    maze_array = np.array([[0 if cell == ' ' else 1 for
cell in row] for row in maze])

    # Mark 'S' and 'G' in the maze array
    maze_array[start[1] // 20, start[0] // 20] = 2  # Use
2 for the start
    maze_array[goal[1] // 20, goal[0] // 20] = 3  # Use 3
for the goal

    # Plot the maze
    plt.imshow(maze_array, cmap=cmap, origin='upper')

    # Extract x and y coordinates from the path and
remove diagonal moves
    path_x, path_y = zip(*path)
    path_x, path_y = remove_diagonal_moves(path_x,
path_y)

    # Plot the path as a red line with adjusted linewidth
    plt.plot(path_x, path_y, color='red', marker='o',
linewidth=0.5)

    # Set labels and title
    plt.xlabel('X-axis')
    plt.ylabel('Y-axis')
    plt.title('A* Pathfinding in Maze')

    # Save the image with the path
    plt.savefig('maze_with_path.png')

    # Show the plot
    plt.show()

# Function to remove diagonal moves from the path
def remove_diagonal_moves(path_x, path_y):
    filtered_x, filtered_y = [path_x[0]], [path_y[0]]

    for i in range(1, len(path_x)):
        # Check if the move is not diagonal
        if path_x[i] == path_x[i-1] or path_y[i] ==
path_y[i-1]:
            filtered_x.append(path_x[i])
            filtered_y.append(path_y[i])

    return filtered_x, filtered_y


# Call this function to visualise the maze with the path
visualize_maze_with_path(maze_array, (start_x, start_y),
(goal_x, goal_y), path)
```

Output:



A* Pathfinding in Maze
0
50
100
150
200
250
300
350
400
Y-axis
0
100
200
300
400
X-axis