Certainly! Here's the code explained with code segments:

```
import cv2 as cv
import numpy as np
import math

def inRange(img, x, y):
    """Check if a point (x, y) is within the image bounds."""
    return 0 <= x < img.shape[0] and 0 <= y < img.shape[1]
```

The `inRange` function checks if a given point `(x, y)` is within the bounds of the image.

```
# Load the image and resize it
image = cv.imread("image_gray2.png")
image = cv.resize(image, (610, 338), interpolation=cv.INTER_LINEAR)
```

Here, the image is loaded using `cv.imread` and then resized to the desired dimensions using `cv.resize`.

```
# Copy the original image to avoid accidental changes
temp = image.copy()
```

The `temp` image is created as a copy of the original image to preserve the original data and avoid accidental changes.

```
# Define color values
blue = [255, 0, 0]
white = [255, 255, 255]
```

```
black = [0, 0, 0]

found = [0, 255, 255]
```

Color values for different elements in the image are defined as RGB arrays.

```
# Define start and end coordinates

startXY = (274, 354)

endXY = (85, 257)
```

The start and end coordinates are defined.

```
# Initialize data structures

open_set = set([startXY])

closed_set = set([])

g = {}

parents = {}

g[startXY] = 0

parents[startXY] = startXY

flag = 0
```

Various data structures are initialized for the A* algorithm. `open_set` stores pixels that have been reached but not explored, `closed_set` stores pixels that have been reached and explored, `g` stores the cost to reach each vertex from the startXY, `parents` stores the parent of each pixel, and `flag` is used to indicate if the optimal path has been found.

```
# Mark the start pixel as explored

temp[startXY] = blue
```

The start pixel is marked as explored by setting its color to blue in the `temp` image.

```

while open_set:

    # Find the node with the minimum cost + heuristic value

    n = min(open_set, key=lambda p: g[p] + math.sqrt((endXY[0] - p[0]) ** 2 + (endXY[1] - p[1]) ** 2))


    # Set the coordinates to n[0] and n[1]

    i, j = n
```

In this while loop, we iterate until the `open_set` is not empty. We find the node with the minimum cost + heuristic value using the `min` function and a lambda function as the key argument. We then extract the coordinates `i` and `j` from the selected node.

```

    # Explore the neighbors

    for r, s in [(0, 1), (1, 0), (1, 1), (-1, 0), (0, -1), (-1, -1), (1, -1), (-1, 1)]:

        if inRange(temp, r+i, s+j):

            if (r+i, s+j) not in open_set and (r+i, s+j) not in closed_set and not np.array_equal(temp[i+r, j+s], black):

                open_set.add((r+i, s+j))

                temp[i+r


, j+s] = blue
```

For each neighbor of the current node, we check if it is within the image bounds using `inRange`. If it is a valid neighbor and has not been explored before, we add it to the `open_set` and mark it as explored by setting its color to blue in the `temp` image.

```

# Once done exploring all the neighbors, remove the current node from the open_set and add it to the closed_set

    open_set.remove(n)

    closed_set.add(n)
```

```
    if flag == 1:
        break
```

After exploring all the neighbors of the current node, we remove the current node from the `open_set` and add it to the `closed_set`. If the flag is set to 1, indicating that the endXY has been reached, we break out of the loop.

```
# Reconstruct the path
if flag == 1:
    reconst_path = []
    n = endXY
    while parents[n] != n:
        reconst_path.append(n)
        n = parents[n]

    reconst_path.append(startXY)
    reconst_path.reverse()

    for i in reconst_path:
        count += 1
        temp[i[0]][i[1]] = found
```

If the flag is set to 1, indicating that the optimal path has been found, we reconstruct the path by backtracking through the `parents` dictionary. We start from the `endXY` and follow the parent pointers until we reach the `startXY`. The reconstructed path is stored in `reconst_path`. We then update the `temp` image to mark the pixels along the path with the `found` color and increment the `count` variable.

```
# Print the path length
print(count)
cv.imshow("Image", temp)
```

cv.waitKey(0)

cv.destroyAllWindows()

```

Finally, the path length is printed, and the `temp` image is displayed using `cv.imshow`. The program waits for a key press (`cv.waitKey(0)`) before closing the image window, and then all the windows are closed using `cv.destroyAllWindows()`.
```