```

import cv2 as cv

import numpy as np

```

The code starts by importing the required libraries: `cv2` for image processing and `numpy` for numerical operations.

```

def inRange(img, x, y):

    """

    Check if a given point (x, y) is within the bounds of the image.


    Args:

        img: The image.

        x: x-coordinate of the point.

        y: y-coordinate of the point.


    Returns:

        True if the point is within the image bounds, False otherwise.

    """

    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. It takes the image, x-coordinate, and y-coordinate as arguments and returns True if the point is within the image bounds and False otherwise.

```

def find_shortest_path(image, startXY, endXY):

    """

    Find the shortest path between two points in a given image using Dijkstra's algorithm.


    Args:
```

```
    image: The input image.
    startXY: Starting point coordinates as (x, y).
    endXY: Ending point coordinates as (x, y).

  Returns:
    The length of the shortest path and an image with the path highlighted.
  """
```

The `find_shortest_path` function finds the shortest path between two points in a given image using Dijkstra's algorithm. It takes the image, starting point coordinates as `startXY`, and ending point coordinates as `endXY` as arguments. It returns the length of the shortest path and an image with the path highlighted.

```
  # Create a copy of the original image to avoid modifying the input image
  temp = image.copy()

  # Define color values for visualization
  blue = [255, 0, 0]
  white = [255, 255, 255]
  black = [0, 0, 0]
  found = [0, 255, 255]
```

A copy of the original image is created to avoid modifying the input image. Color values are defined for visualization purposes. `blue` represents explored areas, `white` is the default color, `black` represents obstacles, and `found` represents the shortest path.

```
  # Initialize sets and dictionaries
  open_set = set([startXY])
  closed_set = set([])
  g = {}
  parents = {}
```

```
    g[startXY] = 0
    parents[startXY] = startXY
```

Several data structures are initialized for the Dijkstra's algorithm implementation. `open_set` stores pixels that have been reached but not explored yet. `closed_set` stores pixels that have been reached and explored. `g` is a dictionary that stores the cost to reach each pixel from the start point. `parents` is a dictionary that stores the parent pixel of each pixel.

```
    # Visualization: mark the explored parts in blue
    temp[startX, startY] = blue

    # Dijkstra's algorithm
    while len(open_set) != 0:
        # Select the node with the minimum cost from the open set
        n = min(open_set, key=lambda p: g[p])

        # Update current coordinates
        i, j = n

        # Explore neighboring pixels
        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):
                # Check if the neighbor has been explored and is traversable
                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
                    parents[(r+i, s+j)] = (i, j)
                    g[(r+i, j+s)] = g[(i, j)] + 1
```

```
            # Check if a shorter path to the neighbor is found
            elif not np.array_equal(temp[i+r, j+s], black):
                if g[(r+i, s+j)] > g[(i, j)] + 1:
                    parents[(r+i, s+j)] = (i, j)
                    g[(r+i, j+s)] = g[(i, j)] + 1

                    # Reopen the neighbor if it was previously closed
                    if (r+i, s+j) in closed_set:
                        closed_set.remove((r+i, s+j))
                        open_set.add((r+i, s+j))

            # Check if the end point is reached
            if (i+r, j+s) == endXY:
                break
```

The Dijkstra's algorithm is implemented using a while loop that runs until the `open_set` is empty. The node with the minimum cost is selected from the `open_set`. The current coordinates are updated, and neighboring pixels are explored. If a neighbor has not been explored, is traversable, and not an obstacle, it is added to the `open_set` and marked as explored in blue. If a shorter path to the neighbor is found, its cost is updated, and if it was previously closed, it is reopened. The loop breaks if the end point is reached.

```
    count = 0

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

```
        reconst_path.append(startXY)
        reconst_path.reverse()

        # Visualization: update the path pixels to 'found' color
        for i in reconst_path:
            count += 1
            temp[i[0]][i[1]] = found

    return count, temp
```

The function reconstructs the shortest path by following the parent pixels from the end point to the start point. The path is stored in the `reconst_path` list. It is then reversed and appended with the start point. The pixels representing the path are updated with the `found` color. The function returns the length of the shortest path and the modified image.

```
# Load the image
image = cv.imread("image_gray2.png")

# Resize the image
image = cv.resize(image, (610, 338), interpolation=cv.INTER_LINEAR)

# Define the start and end points
startXY = (274, 354)
endXY = (85, 257)

# Find the shortest path and get the path length and the image with the path
path_length, path_image = find_shortest_path(image, startXY, endXY)

# Print the path length and display the image with the path
print("Shortest path length:", path_length)
```

```
cv.imshow("Image with Path", path_image)

cv.waitKey(0)

cv.destroyAllWindows()
```

In the main code section, the image is loaded and resized. The start and end points are defined. The `find_shortest_path` function is called to find the shortest path and obtain the path length and the modified image. The path length is printed, and the image with the path is displayed using OpenCV's `imshow` function.