Department of Electrical Engineering

FINAL YEAR PROJECT REPORT

BENGEGU4-INFE-2023/24-GRC-04

Minimum Spanning Forest Finding Algorithm and its Applications

Student Name: Mok Kim Tung
Student ID: 56614570
Supervisor: Prof CHEN, Guanrong
Assessor: Prof CHOW, Tommy W S

Bachelor of Engineering in
Information Engineering

Student Final Year Project Declaration

I have read the student handbook and I understand the meaning of academic dishonesty, in particular plagiarism and collusion. I declare that the work submitted for the final year project does not involve academic dishonesty. I give permission for my final year project work to be electronically scanned and if found to involve academic dishonesty, I am aware of the consequences as stated in the Student Handbook.

Project Title: Minimum Spanning Forest Finding Algorithm and its Applications

Student Name: Mok Kim Tung

Student ID: 56614570

Signature: [illegible]

Date: 12/4/2024

Abstract

The Minimum Spanning Forest (MSF) algorithm is a collection of disjoint trees which connect all the nodes in a graph while minimizing the sum of the edge weight. It can handle applications such as image segmentation by grouping similar pixels and separating pixels according to their differences. Image segmentation is now widely used in object recognition in different fields, such as medical and AI training, and is worthy of investigation[8].

In this project, we studied performing image segmentation with Boruvka's algorithm and compared its performance to the other MST-finding algorithms. We designed a program to generate a matrix containing the intensity levels of each pixel from an image input. To obtain MST, we took the absolute value of the difference between two neighbourhood nodes as the edge weight and applied Borůvka's algorithm. Finally, we process image segmentation and display it in different colors of regions.

The project's core objective is to investigate Borůvka's algorithm's performance in the constructed matrix based on an image input and analyze if its performance is better than the other MST-finding algorithms. Then, the obtained MST list is applied to the image segmentation process with a difference limit variable. Through the simulation, we concluded that Borůvka's algorithm required much more time than others in this type of matrix data converted from an image and the relationship between the variable and the image resolution..

Acknowledgements

I want to express my sincere gratitude to my project supervisor, Professor CHEN Guanrong, for allowing me to participate in this project and for his patience and expertise. His suggestions were crucial in scheduling the process and shaping the research direction and execution of this project. Besides, his recommendations on the research reading materials and his feedback on the progress provided me with foundational knowledge on the topic and inspiration for the project simulation. I am also thankful to my project assessor, Professor CHOW Wai-Shing, for giving feedback on the program simulation and encouraging me during the process.

I appreciate the Electrical Engineering Department of the City University of Hong Kong for providing comprehensive support on the project investigation. The well-planned working schedule facilitated my research. Special thanks to Dr. CHAN Wing Shing for the detailed instructions for the project process and Miss FOK Chiu Yu for the prompt and concise responses to my problems.

I am grateful to my peers for their suggestions on the methodology, constructive criticisms and valuable discussions that contributed significantly to the success of this project. Their opinions help a lot in constructing the simulations.

Lastly, I thank all who directly or indirectly contributed to this project. The experience has been immensely enriching, and I am grateful for the opportunity to work on the simulation and learn.

Use of Generative AI (GenAI) Tools

Use of GenAI tools such as CityU GPT Chatbot are permitted in report writing as long as it is properly cited according to recognized formats such as in APA style. This will be on the condition that you state how and to what extent it is used. This would be its use as a primary source or writing process but not for factual information. You will also need to provide an explanation of why the piece of response from the GenAI tool is adopted.

Contents

PAGE NO.

List of Figures

List of Tables

1. Introduction

The Minimum Spanning Tree (MST) is crucial in graph theory and computational geometry. It is identified as the set of edges within a weighted graph that connects all vertices at the lowest total edge weight possible while avoiding any loop formations. This concept not only forms the backbone of many theoretical models but also boasts practical utility in diverse areas such as computer networking, cluster analysis, and, notably, the segmentation of images within the field of computer vision. The segmentation of an image, which involves dividing a digital image into various segments or pixel groups, is essential for reducing the complexity of an image's representation, thereby rendering it more analyzable and imbuing it with greater significance. Especially in the age of artificial intelligence (AI), image segmentation can help isolate and identify objects within an image. It is a critical approach to process recognitions, medical imaging analysis and applications in multiple fields.

Although numerous algorithms for finding Minimum Spanning Trees (MSTs)—including Kruskal's, Prim's, and Borůvka's—have been widely applied and developed, their application within image data processing has revealed certain constraints. Different MST finding algorithms may perform distinctly on various image data handling methods. Prior research has primarily concentrated on assessing these algorithms' efficiency and computational complexity either in theoretical contexts or through case studies with a limited scope. There is a lack of comparisons between the performance and efficiency of these MST algorithms in image data processing tasks.

1.1 Background

In the project, it uses multiple mathematical structures for the investigation. In this part, we provide the general definitions of the structures and discuss the concepts we will use in this project. We introduce terminology we use in the project to discuss the operation of the three MST finding algorithms (Kruskal's, Prim's and Borůvka's) and image segmentation. We describe weighted undirected graphs, minimum spanning trees and the finding algorithms, and minimum spanning forest.

1.1.1 Weighted Undirected Graph

A weighted graph is a fundamental concept in graph theory in which each edge is assigned a numerical value as the weight of the edge. Directed graphs and Undirected graphs are two classes of weighted graphs. The difference between the two types of graphs is for directed graphs, with arrows indicating the direction of the path. In contrast, for undirected graphs, there are no specific indications for the direction of the path.

For weighted undirected graphs, $G = (V, E, w)$, where:

- V is a set of vertices in the graph. It is non-empty and finite.
- E is a set of edges that can be empty and is also finite. For any $e \in E$ connecting vertices u and v are denoted as an unordered pair$\{u, v\}$.
- $w : E \rightarrow \mathbb{Z}$ is a weight function that assigns an integer to each edge in the graph.

Each edge is said to be incident to a vertex it connects to, and for two vertices that are connected, it is called adjacent. For example, In Figure 1, edge e1 is incident to vertices 1 and 2, while vertex 3 is adjacent to no vertices.

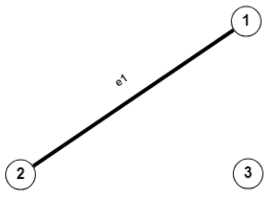


Figure 1 An example graph with two components

There are multiple ways to represent graphs [1]. An adjacency matrix representation can be applied to describe the relations between vertices. The size of adjacency matrix M = (number of the vertices)^2, while the verticle axis denoted source vertice and horizontal axie denoted the destination. For a weighted undirected graph, M must be symmetric along the diagonal. Figure 2 shows an example of a weighted undirected graph $G = (V, E, w)$ where $|V| = 7$, $|E| = 10$ and for example $w(\{A, B\}) = 4$ and $w(\{F, G\}) = 2$. Figure 3 shows the corresponding adjacency matrix representation of the graph described in Figure 2.

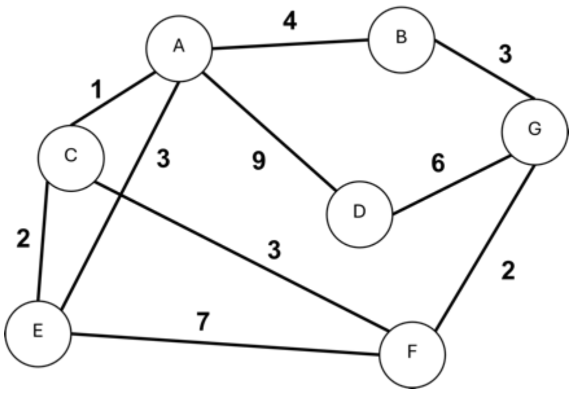
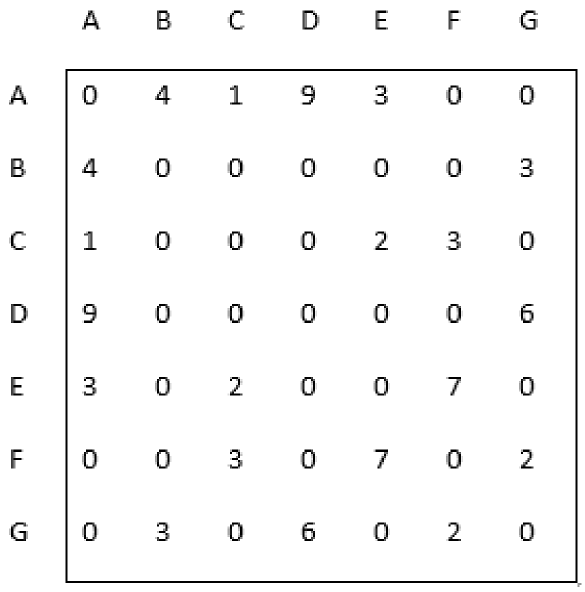


Figure 2: A weighted, undirected graph with 7 vertices and 10 edges



	A	B	C	D	E	F	G
A	0	4	1	9	3	0	0
B	4	0	0	0	0	0	3
C	1	0	0	0	2	3	0
D	9	0	0	0	0	0	6
E	3	0	2	0	0	7	0
F	0	0	3	0	7	0	2
G	0	3	0	6	0	2	0

Figure 3: Corresponding adjacency matrix representation of the weighted undirected graph

1.1.2 Minimum Spanning Trees

A minimum spanning Tree is a fundamental concept in graph theory[2]. Given a connected, undirected graph $G = (V, E)$, a minimum spanning Tree is defined as a spanning tree of G that has the minimum possible total edge weight, including all $v \in V$. The total edge weight W of a spanning tree T can be expressed as:

$$\min W(T) = \min \sum_{e \in T} w(e)$$

There are several properties for minimum spanning tree:

- Multiple minimum spanning trees may be set for the same graph if some edges are of the same weight. Otherwise, if none of the edges share the same weight, the graph's minimum spanning tree is unique.
- The total weight of a minimum spanning tree is the sum of all the edges in the tree, minimized for all the possible spanning trees of the graph.
- For a graph with n vertices, the number of edges in the minimum spanning tree always equals n - 1.

Figure 4 shows a minimum spanning tree $T = \{ E_{A,C}, E_{B,G}, E_{C,E}, E_{C,F}, E_{D,G}, E_{F,G}, \}$ with min $W(T) = 17$, and $|T| = n\text{-}1 = 6$.

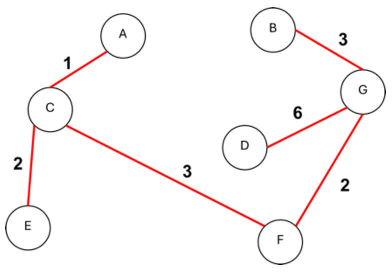


Figure 4: Example of MST from a weighted undirected graph

1.1.2.1 Kruskal’s algorithm

Kruskal’s algorithm is a popular and efficient minimum spanning tree finding algorithm[1]. It is famous for its simplicity and efficiency. It follows a greedy approach, selecting the edges with minimum weight in the graph one by one to ensure they do not form any cycles during the procedure. Here are the steps to perform Kruskal’s algorithm:

I. Arrange all the edges in the graph in non-decreasing order according to the weight.

II. Consider each vertex in the graph as a separate tree, starting with a forest *F* containing all the vertices.

III. Iterate through the sorted list of edges and check if the two vertices the edge incident to are in different trees (Two vertices are not yet connected).

IV. If the vertices are in different trees, add the edge to F and merge the two trees into one. Otherwise, discard the edge if the vertices are in the same tree so as not to form a cycle within F.

V. Repeat the above procedures until there is only one tree in F, which is the graph's minimum spanning tree.

For the time complexity, Kruskal’s algorithm usually takes O(E log E) time to find the minimum spanning tree in the graph.

1.1.2.2 Prim’s algorithm

Prim’s algorithm takes a different approach from Kruskal’s algorithm. It starts with a single vertex and extends the minimum spanning tree by adding the cheapest edge from the tree being built to a vertex that is not yet in the tree[1]. Here is the procedure of Prim’s algorithm for finding the minimum spanning tree in a graph:

I. Start with any vertex in the graph as the initial minimum spanning tree. Mark the starting point as visited.

II. Find the edge with the least weight that connects one of the vertices in the minimum spanning tree and another vertex not in the minimum spanning tree. Mark the vertex found in this step as visited.

III. Repeat the above procedures until all the vertices are included in the minimum spanning tree.

For the time complexity, Prim’s algorithm usually takes O(E log V) time to find the minimum spanning tree in the graph.

1.1.2.3 Borůvka's algorithm

Boruvka's algorithm is one of the oldest algorithms for constructing a minimum spanning from a graph. At the beginning, each vertex is considered a tree. It proceeds in rounds or phases. Each iteration selects the edge with the lowest weight from each tree in the forest, merch the trees together according to the selected edges, and efficiently reduces the number of trees. Each iteration will travel all the trees as one round. Therefore, each round's processing time decreases according to the reduced number of trees until only one tree is left. Here are the steps to perform Borůvka's algorithm:

I. Each vertex in the graph is consider as an individual tree in a forest, starting with one of the trees, the minimum spanning tree is empty. (Figure 5)

II. Travel each tree in the forest, for each tree find and record the edge with minimum weight which incident to target tree and a vertex not included in the target tree. (Figure 6 – Figure 12)

III. Add the selected trees into MST, merge the trees if the selected edge incidents two different trees. During this process, the number of disjoint trees is reduced. (Figure 13)

IV. Repeat the procedure until there is one whole tree left, the minimum spanning tree is constructed. (Figure 14)

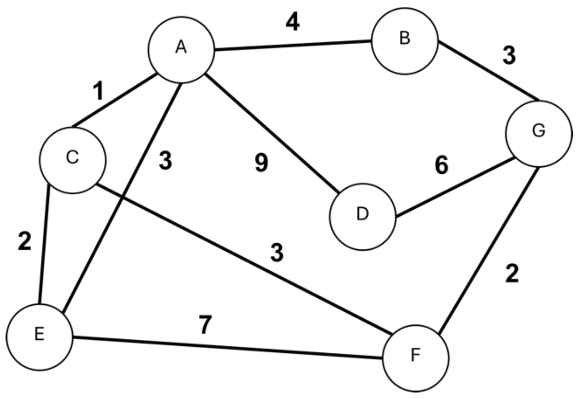


Figure 5: The algorithm accepts a weighted undirected graph as input

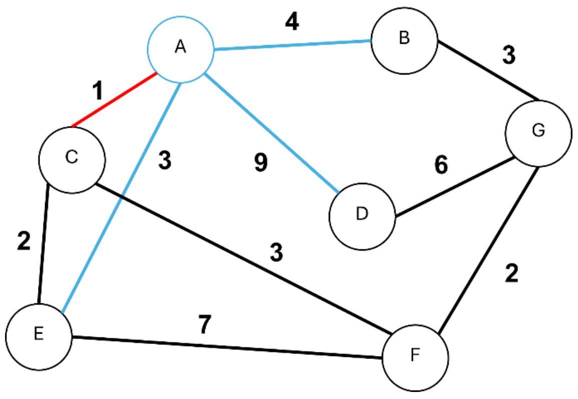


Figure 6: Start from vertex A, check and record the edge with lowest weight

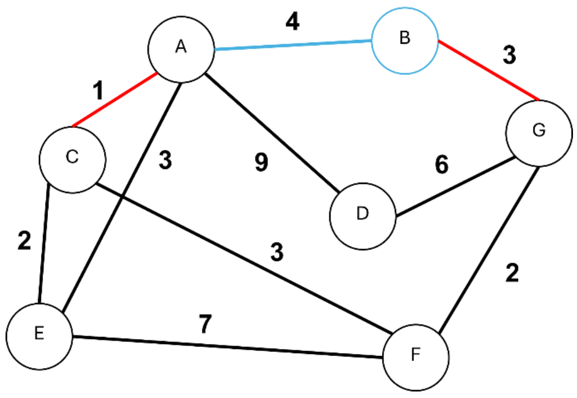


Figure 7: Pass to vertex B, check and record the edge with lowest weight

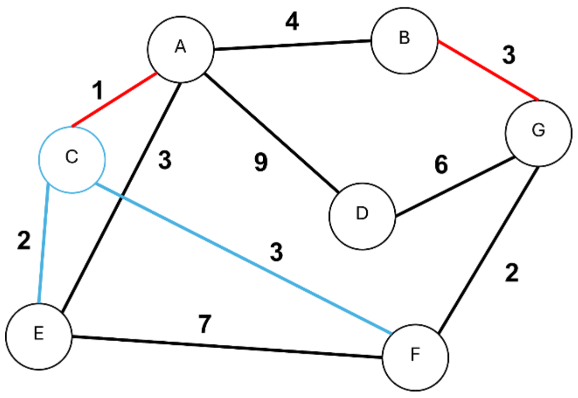


Figure 8: Pass to vertex C, check and record the edge with lowest weight

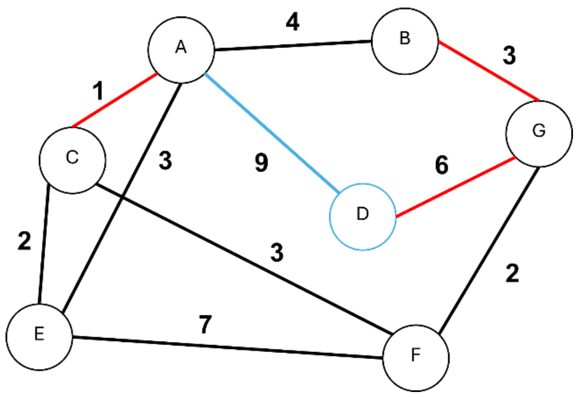


Figure 9: Pass to vertex D, check and record the edge with lowest weight

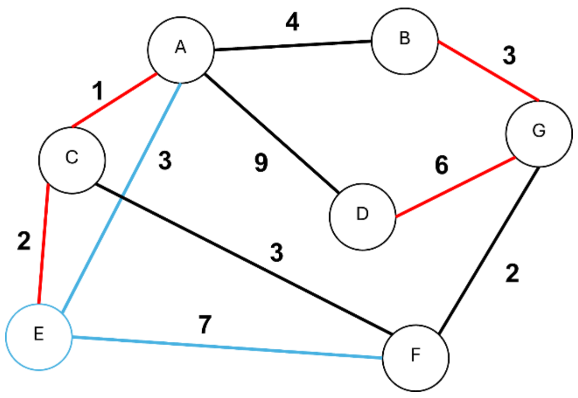


Figure 10: Pass to vertex E, check and record the edge with lowest weight

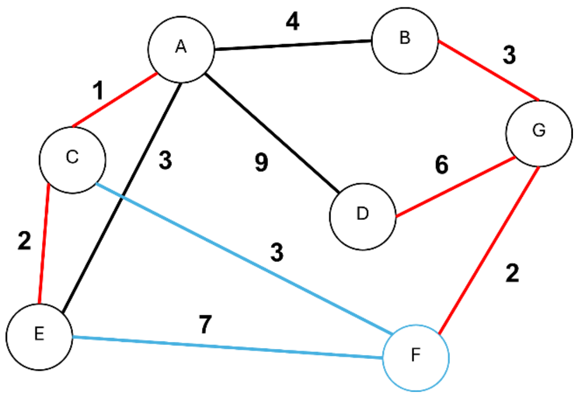


Figure 11: Pass to vertex F, check and record the edge with lowest weight

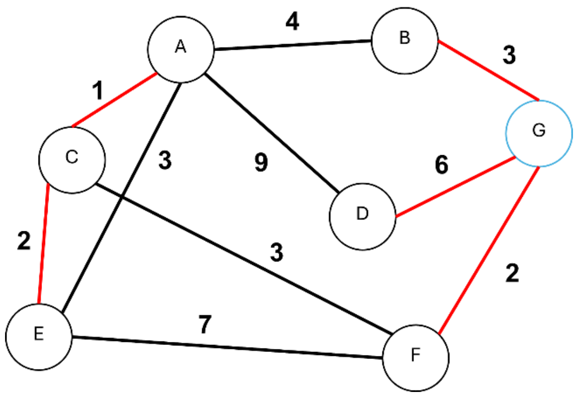


Figure 12: Pass to vertex G, check and record the edge with lowest weight

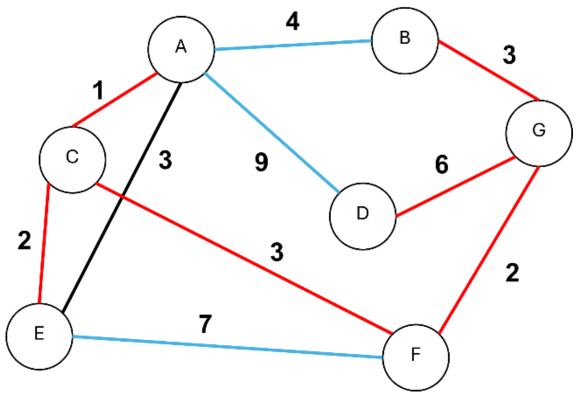


Figure 13: Merge the vertices connected with the recorded edges as a component, start searching and record the edge with lowest weight for all the components

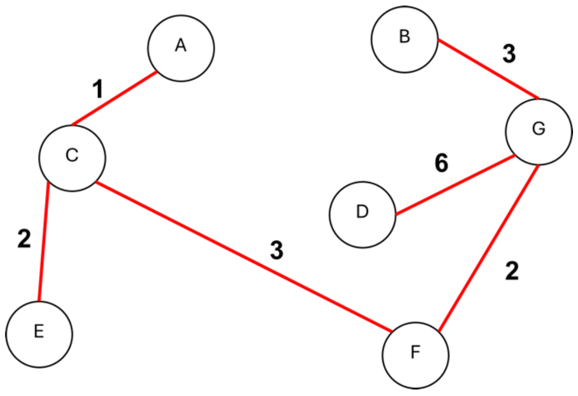


Figure 14: All components are connected and only one component left and out as the minimum spanning tree

For the time complexity, Borůvka's algorithm usually takes O(E log V) time to find the minimum spanning tree in the graph. Besides, as the iteration time decreases according to the reduced number of trees in each round, the process time could be faster according to the graph's structure. For a graph with n nodes, the algorithm could reduce n / 2 (+1 if n is an odd number) components during each iteration for the worst case. For the best case, one round may be enough to construct the minimum spanning tree for the graph.

1.1.3 Minimum Spanning Forest

A Minimum Spanning Forest is a collection of Minimum Spanning Trees that the graph is not necessarily connected, it is a crucial construct in graph theory and finds applications in many domains, such as clustering. Given a graph $G = (V, E, w)$, a Minimum Spanning Forest is defined as a subgraph $F = (V, E')$ where:

- $E' \subseteq E$, and F contains no cycles.
- All $v \in V$ is included in F, but not necessarily connected.
- The total weight of F, given by $\sum_{e \in E'} w(e)$, is minimized all possible subgraphs that satisfy the above conditions.

1.2 Objectives

This study aims to compare and investigate the performances of three MST finding algorithms on a classic adjacency matrix generated from different sizes of images and further process image segmentation with Borůvka's algorithm-based function. The project has two main goals:

I. Compare the efficiency of finding a minimum spanning tree from a graph between Borůvka's algorithm and the other two minimum spanning tree finding algorithms (Kruskal's and Prim's) and analyze the obtained result.
II. Further process image segmentation with the minimum spanning tree list obtained in target one and analyze the result generated by different parameter values.

We recorded the pixel intensity level from the greyscale image to achieve the target I and constructed an adjacency matrix as an input to the minimum spanning tree-finding algorithms. After obtaining the processing time of each algorithm, we analyze the relationship between the input matrix size and the processing time to compare the performance of the algorithms.

For the second target, we will then extract the minimum spanning forest list from the minimum spanning tree obtained in the last part and edit the pixels included in each tree in the minimum spanning forest list, which is not connected.

2. Methodology

This is a simulation-based project. We choose Jupyter Notebook as the programming platform and Python 3 as the programming language. To carry out the information we need to analyze the result, we defined various functions to handle the input image resources and minimum spanning tree list obtained from the algorithms.

We divided the complete process into six parts to introduce the concepts and functions of each part of the program. Below are the descriptions of each part, which explains what the section will take as input and what to carry out:

2.1 Import Libraries

We selected various libraries to assist in handling the data generated during the process. Also, to handle image data, we imported some libraries to handle image processing and preview the image output.

2.1.1 OpenCV

OpenCV is an open-source library that includes several hundreds of computer vision algorithms[7]. In this project, we apply OpenCV functions mainly to image processing, including getting and storing the feature from the input image, image resizing, and image pixel editing.

2.1.2 NumPy

NumPy is an open-source Python library that is used in almost every field of science and engineering[6]. In this project, we use the NumPy array functions to handle matrices in large sizes. As the size of the NumPy array is according to the memory on the machine, we can handle much more extensive data with the NumPy array instead of the ordinary list function from Python.

2.1.3 heapq (Heap Queue Algorithm)

This module provides an implementation of the heap queue algorithm. In this project, heapq functions are only used to construct Prim’s minimum spanning tree algorithms. Heapq can help reducing the function length of Prim's algorithm.

2.1.4 random

This Python built-in function generates random numbers from a variable range. This project will separate different regions from the image using different colors. Therefore, we decided to apply random numbers from 0 to 255 on three different variables, representing red, green, and blue color values in a pixel. In this way, the possibility of the function getting the same color in different iterations is very low $(1 / 256)^3$.

2.1.5 Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. In this project, we decided to preview the image using the current process instead of a pop-up window. Therefore, we import pyplot from matplotlib and apply it to the coding block to generate the desired image output type. As a result, it is easier for us to track the image situation and debug the functions if there are some logical errors.

2.2 Input Image and Preparations

We will need only one image resource as input in this simulation program. During this section, we worked on the preparations for the input image, including resizing the image and getting the required information from the image.

We can read an image file with OpenCV and get the information we need for other sections. This project only considers image files in .jpg and .png format. In this first step, we input the image file with cv2.imread(<image file path>, 0) to turn the image into greyscale and store the NumPy array generated from the image into a variable. Figure 15 shows an example image with size 599 * 567 pixels to be imported into the program, and the image was converted into greyscale with the same size as shown in Figure 16.

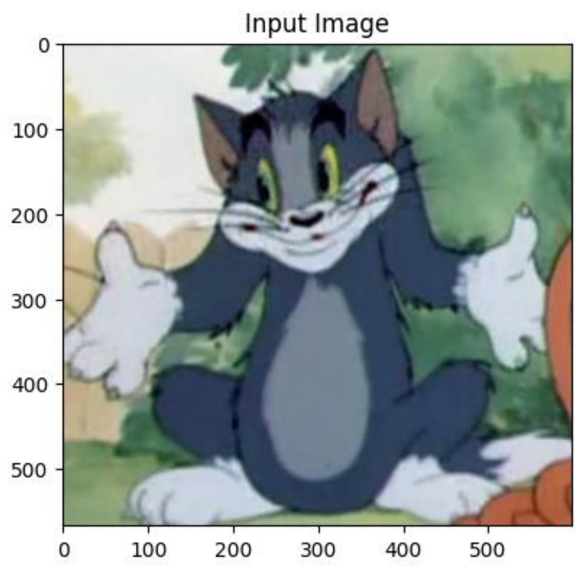


Figure 15: Shrugging Tom [Digital image]. (2020). Retrieved from https://knowyourmeme.com/photos/1722871-shrugging-tom

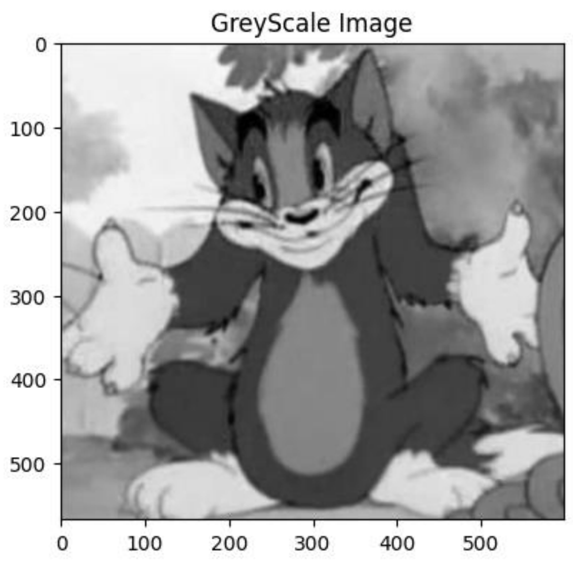
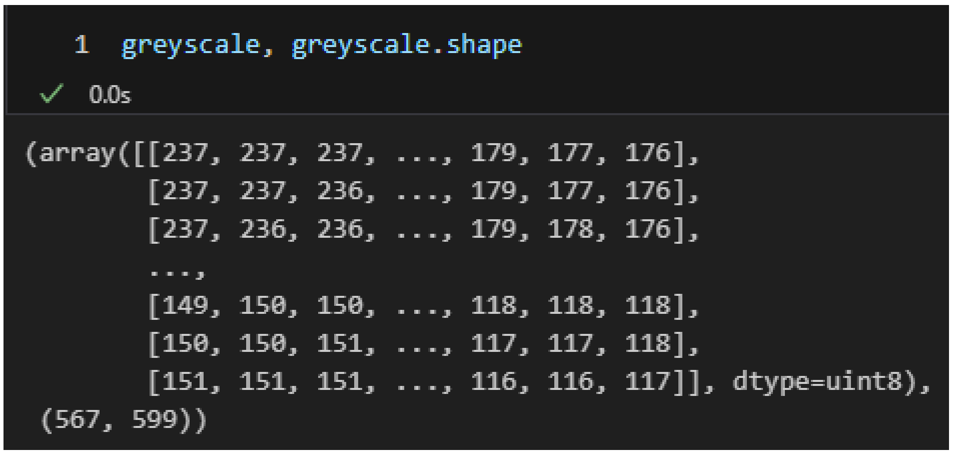


Figure 16: Greyscale image with the same resolution transformed from the same meme image.

Figure 17 shows that the NumPy array currently stores each pixel's pixel intensity level. The value of the pixel intensity level is according to the pixel's brightness. The closer the value pixel intensity is to zero, the darker the pixel's visual color. On the contrary, the closer the value pixel intensity level to 255, the brighter the visual color of the pixel. It is under the same concept of adjusting the RGB levels on the monitor: If we set all the values to zero, the visual display is too dark to read the contents on the screen; If we set all the values to 255, then it may be too bright to read the contents on the screen.



```
greyscale, greyscale.shape
```

```
✓ 0.0s
```

```
(array([[237, 237, 237, ..., 179, 177, 176],
        [237, 237, 236, ..., 179, 177, 176],
        [237, 236, 236, ..., 179, 178, 176],
        ...,
        [149, 150, 150, ..., 118, 118, 118],
        [150, 150, 151, ..., 117, 117, 118],
        [151, 151, 151, ..., 116, 116, 117]], dtype=uint8),
 (567, 599))
```

Figure 17: A 2D NumPy array with the label *resizedImg* and resolution 599 * 567 storing the pixel intensity level of all the pixels in the greyscale image

Besides, we choose to turn the image source from BRG (default image color arrangement under OpenCV) to greyscale because this project aims to test the efficiency of the minimum spanning tree algorithms. Therefore, the greyscale image can reduce the information from three layers (B, R, G) to one layer (Pixel intensity level), and it would be enough for us to observe the result.

After storing the image data with the NumPy array, we can modify the array for further processes. Figure 18 shows that the greyscale image is resized to a smaller resolution to reduce the number of pixels. By editing the resizeScale parameter, the input image file's width and height are reduced in a percentage and stored in other variables. Also, the height, width, and the number of pixels are equal to the product of height and width. Figure 19 shows variables and a new NumPy array to store the information of the resized greyscale image.

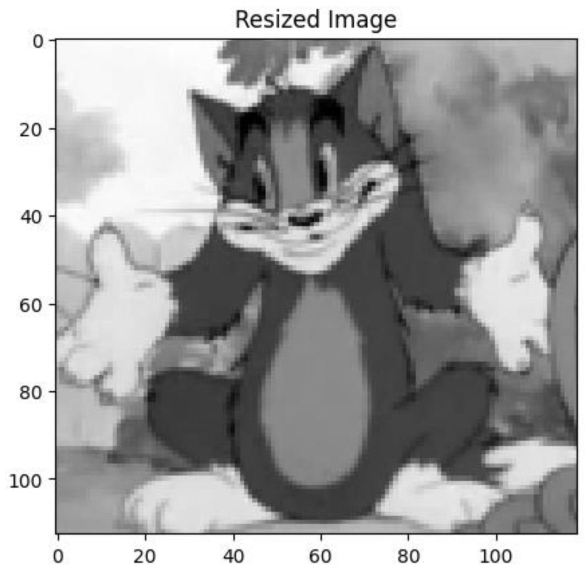
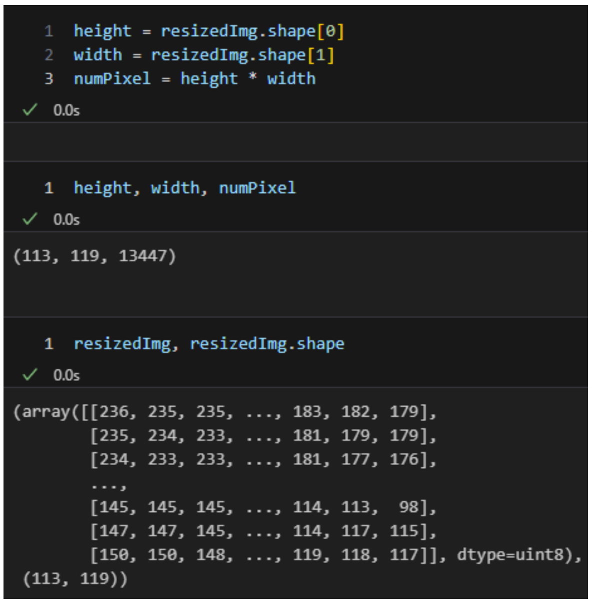


Figure 18: A resized greyscale image with the resize scale equal to 20%



```
height = resizedImg.shape[0]
width = resizedImg.shape[1]
numPixel = height * width
```
✓ 0.0s

```
height, width, numPixel
```
✓ 0.0s

```
(113, 119, 13447)
```

```
resizedImg, resizedImg.shape
```
✓ 0.0s

```
(array([[236, 235, 235, ..., 183, 182, 179],
        [235, 234, 233, ..., 181, 179, 179],
        [234, 233, 233, ..., 181, 177, 176],
        ...,
        [145, 145, 145, ..., 114, 113,  98],
        [147, 147, 145, ..., 114, 117, 115],
        [150, 150, 148, ..., 119, 118, 117]], dtype=uint8),
 (113, 119))
```

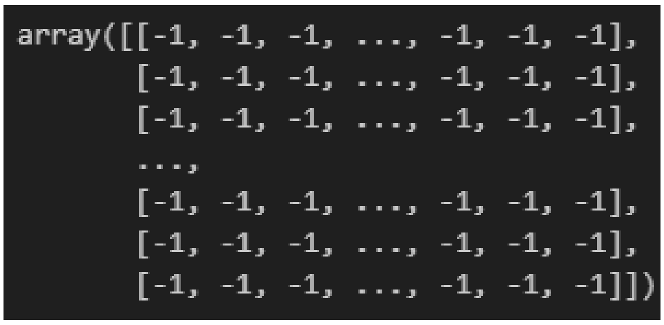
Figure 19: Variables and NumPy array storing the information of the resized image

2.3 Matrix Construction

To find the minimum spanning tree, we need a weighted undirected graph and the adjacency matrix representing the relationship between vertices by denoting the weight of the edges between vertex u and vertex v in Matrix[u][v]. To construct the matrix, we consider each pixel in the image as the vertices of the weighted undirected graph. However, for the image, there are no edges between the pixels for the algorithm to consider the edges with the lowest weight. Therefore, we made an edge weight for each by calculating the absolute value of the difference in the pixel intensity level of the adjacent pixels.

2.3.1 NumPy Array to Store the Adjacency matrix

This part is to create a 2D NumPy integer array and fill all the elements with -1. The reason for choosing -1 instead of 0 to declare that the vertices are not adjacent is that some adjacent pixels may have the same pixel intensity level. In this case, the edge weight, equal to the pixel intensity level difference between two vertices, is zero. Therefore, we consider zero as a valid input that the pixel intensity level of two pixels is the same instead of not adjacent to each other. The matrix with side length 13447, equal to the number of pixels in the resized image, is initialized with filling -1, as shown in Figure 20. For Matrix[*u*][*v*], which denoted the weight of *Edgeu,v*, u represented the source vertex on the y-axis of the matrix; and v represented the destination vertex on the x-axis of the matrix.



```
array([[-1, -1, -1, ..., -1, -1, -1],
       [-1, -1, -1, ..., -1, -1, -1],
       [-1, -1, -1, ..., -1, -1, -1],
       ...,
       [-1, -1, -1, ..., -1, -1, -1],
       [-1, -1, -1, ..., -1, -1, -1],
       [-1, -1, -1, ..., -1, -1, -1]])
```

Figure 20: A 2D NumPy array storing -1 as the initial value with the matrix length equal to number of pixels in the resized image

2.3.2 Calculating the Edge Weight for Each Vertex to Adjacent Vertices

In this part, we designed four functions to calculate the edges for each vertex. Each vertex may have at least three edges and, at most, eight edges connecting with the adjacent vertex. However, some of the edges repeat for different vertex. For example, figure 21 shows all the outgoing edges from A to other adjacent vertices; Figure 22 shows all the outgoing edges from B to adjacent vertices. For vertices A and B, $Edge_{A,B}$ and $Edge_{B,A}$ is the same edge with the same weight 4. Therefore, we decided on four functions representing four directions for each vertex to calculate and record their outgoing edge weight instead of eight directions. In this way, the processing time needed to construct the adjacency matrix could be reduced.

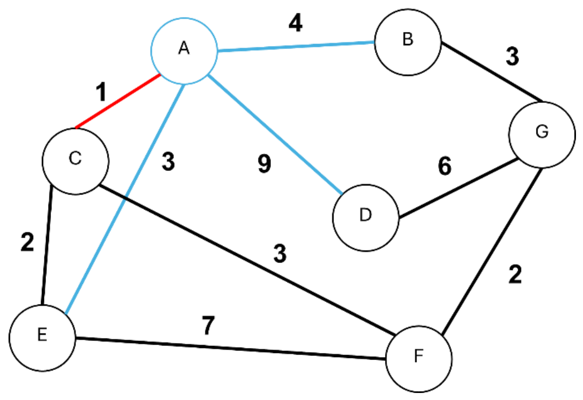


Figure 21: Outgoing edges from vertex A

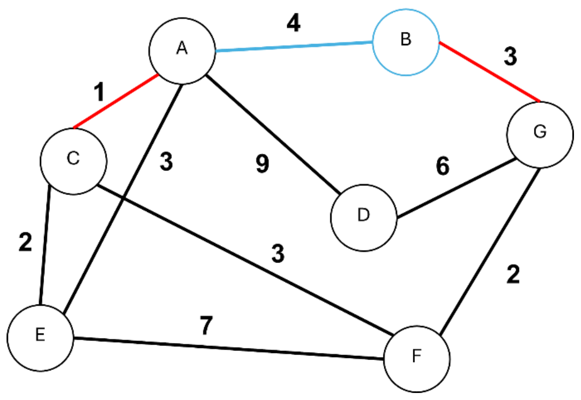


Figure 22: Outgoing edges from vertex B

The edge-building functions will loop from pixel[0][0] to pixel[height][width] until all the pixels are processed. The sequence in detail first adds the width index to the limit of the image's width. Once the width index reaches the limit, add one to the height index. When both indices reach the limit, the function has checked all the pixels and the looping ends. For example, Figure 23 shows a two times three matrix representing an image of the same size. The sequence for the function to check the pixel is from [0][0] to [0][1], [0][2]. Once it reached the end of the row, the height index added one and kept running the function until it reached [1][2].

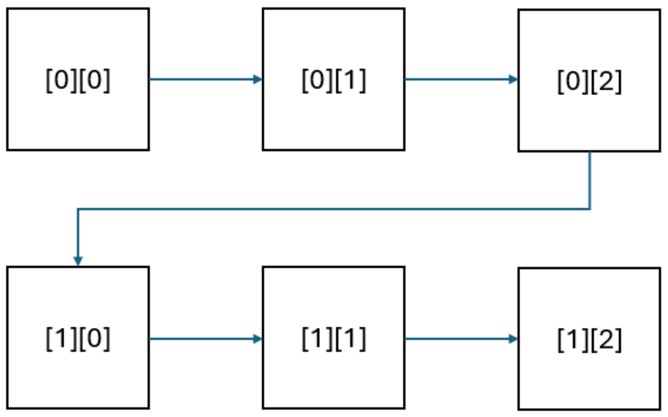


Figure 23: Function process sequence on a 2*3 image example

For example, the four selected directions are the bottom left, bottom right, and right sides of the source vertex. Figure 24 shows the same structure of the matrix as Figure 23; we start with the pixel with indices [0][0] and consider where functions should be checked to construct a weighted edge for the adjacent vertices. We can see that for pixel[0][0], there are

3 adjacent vertices including pixel[0][1], pixel[1][0] and pixel[1][1]. Therefore, we should apply three functions to check for three directions and construct the weighted edge.

Next, the function should check pixel[0][1] by following the abovementioned sequence. For this pixel, as shown in Figure 25 there are 5 adjacent vertices, including pixel[0][0], pixel[1][0], pixel[1][1], pixel[1][2] and pixel[0][2]. However, from the previous loop, we have already considered the edge incident to pixel[0][0] and pixel[0][1]. Therefore, four directions from pixel[0][1] are required to be checked.

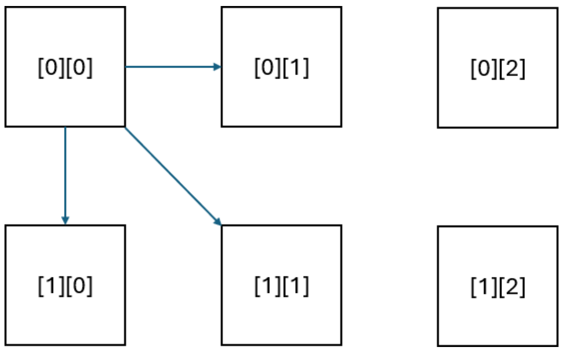


Figure 24: Example of checking targets from pixel[0][0]

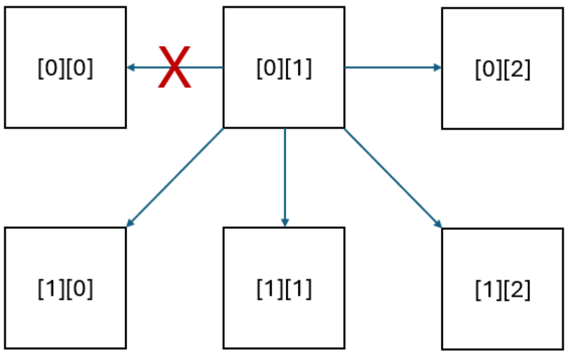


Figure 25: Example of checking targets from pixel[0][1]

Following the above logic, we can set up rules to let the program decide which directions should be checked for each pixel in the image. The rules are listed below. Pixels are denoted by *P*[*y*][*x*], in which *y* is the position on the y-axis and *x* is the position on the x-axis:

I. If x > 0 and y < image height – 1, then check for bottom left side and construct a weighted edge.
II. If y < image height – 1, then check for bottom side and construct a weighted edge.
III. If x < image width – 1 and y < image height -1, then check for bottom right side and construct a weighted edge.
IV. If x < image width – 1, then check for right side and construct a weighted edge.

With the above rules, we can obtain a matrix *M* (*y*, *x*) with below properties:

- Matrix size is the product of height and width of the image.
- First index y denoted the source vertex, and Second index x denoted the destination index.

- $M[y][x]$ denoted the edge weight of the edge connecting vertex x and vertex y.
- Cutting the matrix with a diagonal from top left to bottom right, only the upper part, which is half of the matrix is modified. For the further processing, only the upper part is used for calculations.

To calculate the number of the edges, we can consider the number of edges generated by the functions adding edges in four directions. For an image I with H as height and W as width, and the number of pixels N equals to $H * W$:

- Apply rule I, there are $(H – 1) * (W – 1)$ vertices matched the rule and generated equal numbers of edges.
- Apply rule II, there are $(H - 1) * W$ vertices matched the rule and generated equal numbers of edges.
- Apply rule III, there are $(H - 1) * (W – 1)$ vertices matched the rule and generated equal numbers of edges.
- Apply rule IV, there are $H * (W - 1)$ vertices matched the rule and generated equal numbers of edges.

To summarise the procedures, the sum of the edges generated is:

$(H – 1) * (W – 1) + (H - 1) * W + (H - 1) * (W – 1) + H * (W - 1)$

$= 2*(H * W – H – W + 1) + 2(H * W) – H – W$

$= 4 * (H * W) – 3 * (H + W) + 2$

2.4 Minimum Spanning Tree Finding Algorithms

In this part, we defined the three minimum spanning tree finding algorithms (Kruskal's, Prim's and Borůvka's) referencing the procedure of each algorithm. The primary purpose of this part is to find the minimum spanning tree from the adjacency matrix with different minimum spanning tree finding algorithms and compare it with the efficiency of gaining the result. Below is the logic flow of each minimum spanning tree finding algorithm with pseudocodes.

2.4.1 Kruskal’s Algorithm

```
def kuskal(adjacency matrix):
    n = number of vertices
    edges = [all edges in the upper part of the adjacency matrix with their weight]
    edges.sort(based on the weight of the edges)
    parent = [for i in range(matrix side length)]
    rank = 0 * matrix side length
    MST = []
    for source vertex, destination vertex, weight in edges
        find roots of the two vertices with helper functions
    if u and b having different roots:
        add edge into MST
        merge the two components u and b belongs to
return MST
```

2.4.2 Prim’s Algorithm

```
def prim(adjacency matrix):
        n = number of vertices
        inf = infinity
        key = initialize all keys of vertices as inf
        parent = initialize all parent indices of vertices as -1
        in_mst = initialize false for all the vertices as none of them in the MST
        priority queue = [start from (0, 0)]
        key[0] = 0
        while pq is not empty:
         pop vertex with smallest key from pq
         if in_mst[vertex] = True:
                 skip
         in_mst[vertex] = True
         for next vertex in range(n):
                 if vertex < next vertex:
                         weight = adjacency matrix[vertex][next vertex]
                 else:
                         weight = adjacency matrix[next vertex][ vertex]
                 if weight not smaller than 0 and next vertex is not in MST and key[next
vertex] > weight:
                         key[next vertex] = weight
                         parent[next vertex] = current vertex
                         add next vertex to pq
        MST = (parent vertex, vertex, weight) in range(from 1 to n) and parent vertex != -1

return MST
```

2.4.3 Borůvka's algorithm

```
def boruvka(adjacency matrix):
    n = number of vertices
    parent = [i for i in range(n)]
    rank = [0] * n
    MST = []
    numTrees = n
    While numTrees larger than 1:
        cheapest = [initialize cheapest edge for each tree with (source, destination,
weight)]
    for i in range(n):
        for j in range(from i+1 to n):
            weight = adjacency matrix[i][j]
            if weight larger than -1
                find the root of vertex i and j
                if two root are different:
                    check both cheapest edge weight.
                    if root cheapest weight > weight:
                        update the root with new root and weight
    for i in range(n):
        source, destination, weight = cheapest[i]
        if source exist and roots of source and destination are different:
            add edge to MST
            merge two component into one
            numTrees reduces by 1
    if all source vertex of edge in cheapest list equal to -1
        exit
return MST
```

2.4 Checking

After defining the minimum spanning tree finding algorithms, we obtained three sets of sorted minimum spanning tree lists by inputting the same adjacency matrix to the functions in the format of [(source vertex, destination vertex, weight) …].

To verify the accuracy of the list, we defined two functions to check:

I. The minimum spanning tree list has included all the vertices in the graph.
II. The minimum spanning trees are in the same minimum weight.

For the first point, the concept of the function is to detect the properties of the input minimum spanning tree list. It will return False under multiple situations:

- As the number of edges for a minimum spanning tree with n nodes is always n – 1, if the length of the list is not equal to n – 1, it is not a minimum spanning tree.
- Check for each edge included in the list, if there are two edges sharing same root, it declared that a cycle is existed. Therefore, it is not a minimum spanning tree.

For the second point, we take the result generated by Kruskal's algorithm as the criteria to check if the generated minimum spanning tree list is the same with the same total weight. If the list failed the previous checking or does not have the same weight as the criteria, it would return False to declare the generated result is not the minimum spanning tree list.

2.5 Extract Trees from MST to Construct MSF

One of the approaches to process image segmentation is grouping up pixels with low differences. We designed a function to extract trees from the input minimum spanning tree list and form a minimum spanning forest in which the trees in the list are disjointed according to the difference of the edges used to incident two vertices.

The function's concept is to check every single edge included in the minimum spanning tree list. It started with the first edge by adding the edge from the minimum spanning tree list to a new list, usually denoted as (0, 1, weight) and removed it from the minimum spanning tree list; then, it checked for other edges which are outgoing edges from the source vertex and destination vertex of the vertex checking currently. If the absolute value of the difference between the weight of two edges is lower than an input integer, the function will consider that two edges should be in the same component. Like the previous edge, the current edge will be appended to the new list and removed from the old minimum spanning tree list as the two components merge into one component. The function will stop until the minimum spanning tree list is empty, meaning all the edges were checked and merged with the same edges. At last, it will return a 2D list as MSF[trees][nodes in the tree]. Below is the pseudocode of the function:

```
def extractMSF(MST list, target difference):
    MSF = empty 2D list
    processed edges = set() to avoid duplication
    for i, current edge checking in enumerate(MST list):
        if i is processed:
            skip
        current vertex group = set([source and destination vertex of current edge]])
        stack = [current edge checking]
    while stack is not empty:
        pop the edge from stack and store as edge
        for j, other edge in enumerate(MST list):
            if j is not processed and edge is connected with other edge:
                update current vertex group with source and destination
vertex of other edge
                add other edge to stack for next iteration
                mark j as processed
    append current vertex group to MSF
return MSF
```

2.6 Image Segmentation with the Constructed MSF List

In the last step, we segmented the image with the minimum spanning forest list by giving a random RGB value to the pixels in the same tree. We first turned the greyscale image to RGB to change the color of the pixels. If this step is missed, each pixel can only contain the pixel intensity level, and it will be hard to separate different regions. Then, we set up a for-loop to generate three random values for red, green, and blue colors from 0 to 255, respectively. In this case, the probability of obtaining the same values for three colors should be (1 / 256)3 = 1 / 16777216, which is rare. As last, we converted the vertex number back to the coordinate of the pixel located in the image matrix. For example, Figure 26 shows a 2 * 3 matrix M1, which represents an image, denoted by the vertices with the coordinate, i.e. M1[y][x], and Figure 27 denoted the vertices in the same matrix with sequence number N obtained from the formula sequence N = y * width of image + x. Taking M1[1][2] as an example, N = 1 * 3 + 2 = 5, equal to the value shown in Figure 27.

- To reverse the sequence number back to coordinate, we applied the below formulas:y = integer(N // image width), and
- x = integer(N % image width)

We can get the y and x values from the sequence values by applying these formulas. Take vertex 4 as an example. y = integer(4 // 3) = 1, and x = integer(4 % 3) = 1. Then, we can obtain the coordinate of vertex 4 located in the image matrix.

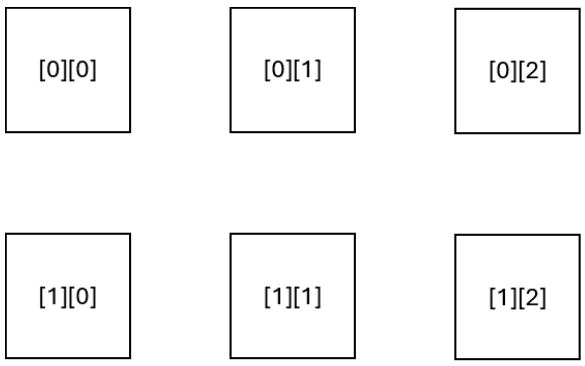


Figure 26: Example 2*3 Matrix denotes the vertices with coordinate

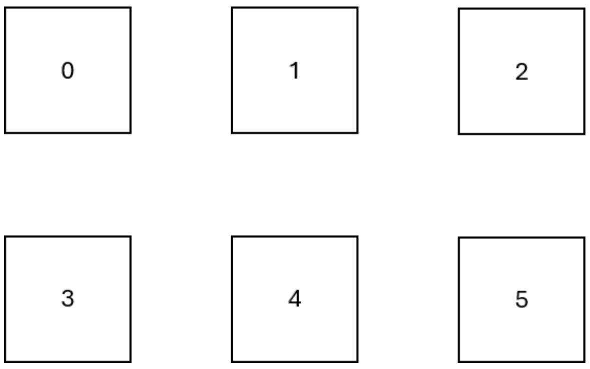


Figure 27: Example 2*3 Matrix denotes the vertices with sequence

3. Results & Discussion

In this section, we will mainly display the results observed and analyzed from two program parts. The first part will focus on minimum spanning tree constructions with different algorithms. The primary purpose is to show and explain the results from constructing the input resources to the efficiency of the minimum spanning tree-finding algorithms. The second part aims to show the segmentation results on different values of variables, which consider the connections between vertices in the image.

3.1 Comparison Between the Performances of the Minimum Spanning Tree Finding Algorithms

To test the functions' performance for finding the minimum spanning tree, we chose three images in different resolutions, equal to the total number of pixels included, to test the processing time in each function. Below is the table listing the performances of the functions among three different resolutions of image input:

Table 1

Information and performances on processing functions of different resolution of image input

Image resolution (Pixels)	56*59	170*179	283*299
Number of Pixels in the image	3304	30430	84617
Number of Edges between Pixels	12873	120675	336724
Size of the Adjacency matrix	10916416	925984900	7160036689
Adjacency matrix Construction (sec)	0.0	0.5	29.3
Kruskal's Algorithm (sec)	0.7	59.5	491.4
Prim's Algorithm (sec)	1.6	151.3	1830.7
Borůvka's algorithm (sec)	4.5	427.9	3670.3

Table 1 described the data of the program collected during the processing stage. We can summarise the following points:

3.1.1 Image Scaling and Matrix Construction

According to the increased number of input image files, the number of vertices and edges generated from the vertices was increased in a similar ratio. For example, the image in resolution 56 * 59 (size 1) contained 3304 pixels in the file, and the image in resolution 170 * 179 (size 2) contained 336724 pixels in the image. With the formula, we obtained to calculate the number of edges generated from the function; for image resolution 1, the number of edges = 4 * 56 * 59 – 3 * (56 + 59) + 2 = 12873 edges, which is the same as we obtained from the simulation. This declares that our estimation of the calculation of the edge is correct. However, an image with the same number of pixels and different image height and width values could generate different edges. For example, an image with resolution 118 * 28 has the same number of pixels as size 1, but it will generate 12780 edges after running the edge generator functions. Therefore, we can summarise the number of edges to be produced as only related to the height and width value of the image instead of the number of pixels.

Then, using the same method, the number of edges for image resolution 2 is 120675. With the above value, we can obtain the ratio percentage of the number of pixels and the number of edges between two image resolutions, which is 3304 / 30430 = 10.86% and 12873 / 120675 = 10.67%. Besides, the resizing scale of size 2 into size 1 is (56 / 170) * (59 / 179) = 9.89%, which is very close to the ratio percentages we calculated in the previous steps. Again, applying the formula in size 3, 336724 edges were produced during the process, and the ratio of the number of pixels and the number of edges between size 1 and size 3 is 3304 / 84617 = 3.9% and 12873 / 336724 3.8%. For the resizing scale, (56 / 283) * (59 / 299) = 3.9%. With the two comparisons, we can summarise that the number of vertices and edges decreases according to the resize scale for images having the same ratio of height/width. Last, on the scaling problem, the side of the adjacency matrix stores all the edges, and the scaling ratio is the square of the scaling ratio of the image files.

3.1.2 Efficiency on Borůvka's Algorithm against to the others

From the processing time collected, it is evident that the processing time for Borůvka's algorithm (B) to find the minimum spanning tree is much longer than Kruskal's algorithm and Prim's algorithm. Comparing its performance against Kruskal's algorithm (K) and Prim's algorithm (P) in three different size image inputs:For an image with resolution 56 * 59, B required 4.5 seconds to perform the task, which is about 6.4 times and 2.8 times of the time required for K and P respectively.

- For an image with resolution 170 * 179, B required 427.9 seconds to perform the task, which is about 7.19 times and 2.8 times of the time required for K and P respectively.
- For an image with resolution 283 * 299, B required 3670.3 seconds to perform the task, which is about 7.46 times and 2 times of the time required for K and P respectively.

The comparisons show that both the ratios of the processing time between B and K and P could be described as changing oppositely. Analyse the result of B to K; the processing time ratio increases according to the increase in the matrix size. Besides, comparing B to P, the processing time ratio decreases from 2.8 to 2.

For K, the overall processing time to get the minimum spanning tree is usually denoted as E log E, mainly related to the number of edges in the graph.

Next, for P, the overall processing time is usually denoted as V2 log V when applying an adjacency matrix and binary min-heap in the function. Besides, for B, the overall processing time is denoted as E log V. With the above information, we can estimate the priority of which algorithm is taking less time to process the minimum spanning tree. In this study, the number of generated edges during the process is about four times the number of vertices, i.e. E = 4V.

Therefore, the processing time estimations are 4V log 4V, V2 log V, and 4V log V for K, P and B, respectively. In this situation, it is easy to rank the algorithms according to the required processing time in ascending order: B à P à K for V not larger than 4. However, it is the opposite of the result we obtained from the program. There are some possible causing B to take longer time in the simulation:

1. Function Structure Design

   The time complexity of the algorithms denoted the upper bound of the processing time in the worst case. Besides the function structure, it is also a factor of why the function takes extra time to process a result.

2. Data Structure

   In this study, the input adjacency matrix is transferred from the image by calculating the difference between the pixel intensity levels of adjacent vertices. In this case, the weight of the edge could be zero for multiple components in the matrix. This may affect the processing efficiency of the minimum spanning tree-finding algorithms. For example, B may need more time to compare with the edges as multiple edges from the same source vertex have the same weight.

3. Number of components

   The larger the image's resolution, the larger the number of edges to be considered. In each round of iteration in B, it is required to check all the edges connected to each component in every round. Initially, each vertex is considered as a small component. When the number of components is huge, each component has a significant number of comparisons. Although the number of components will decrease in each round, comprehensive checks across all the edges will still be required multiple times in the early stage.

   Besides, K sorts all the edges only at the beginning, then iterates through the sorted list and works on a union-find check for each edge to determine if it is included in the minimum spanning tree list. Moreover, P uses a priority queue to efficiently manage and update the next most favourable edge to be added to the minimum spanning tree list. As a result, in this situation, B's performance could be worse than that of K and P.

3.2 Comparison Between the Performances of Image Segmentation under Different Edge Weight Limit

In this section, we input the generated minimum spanning tree list to output a minimum spanning forest list by splitting the trees from the minimum spanning tree from the edge with an edge weight too high to consider the components connected to one region. To experiment with the relationship between the weight limit for the edges to be split and the resolution of the image, we process image segmentation on the three different resolutions of the image and obtain the result with different values of the weight limit variable to generate visible image results for observation and analysis. Below is the table showing the processing time of image segmentation on different resolutions of images and image output from applying image segmentation functions based on three levels of edge weight limit on three different resolutions of image input.

Table 2

Performances on processing image segmentation of different resolution of image input

Image resolution (Pixels)	56*59	170*179	283*299
Extract MSF from MST (sec)	1.0	81.8	690.1
Image Segmentation (sec)	0.2	0.2	0.3

Comparing the processing time, we can observe that the time required to get a list storing trees with a difference in pixel intensity is lower than a set variable changes positively according to the image's resolution. Besides, the time required to change the pixel's colour according to the minimum spanning forest list does not change according to the image resolution.

In the simulation part, we take images in resolution 56 * 59 (size 1), resolution 170 * 179 (size 2) and resolution 283 * 299 (size 3) as the image input and apply the weight limit values 3, 5 and 7 for each resolution of image input. Below is the image output after processing the image segmentation function, based on different values as the weight limit to filter the approach edges:

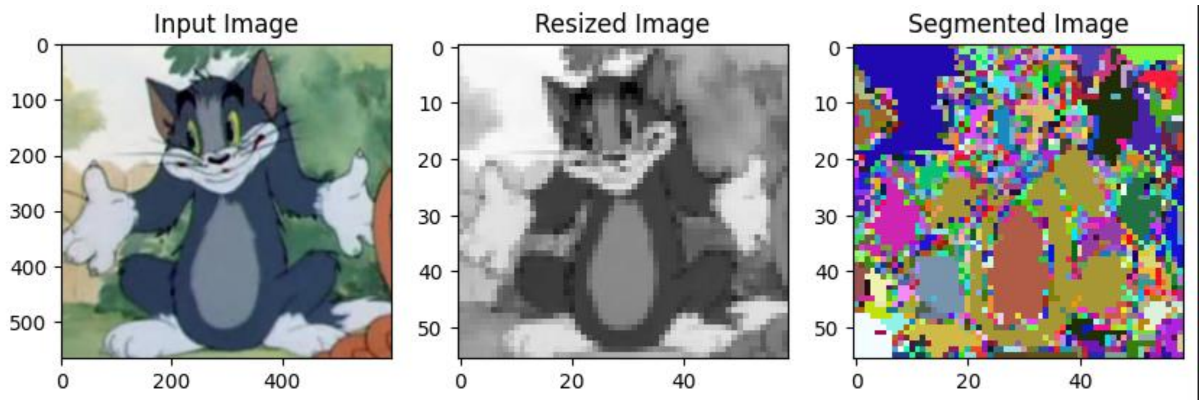


Figure 28: Original input image, resized greyscale image and segmented image with edge weight limit as 3 with image resolution of 56*59 pixels

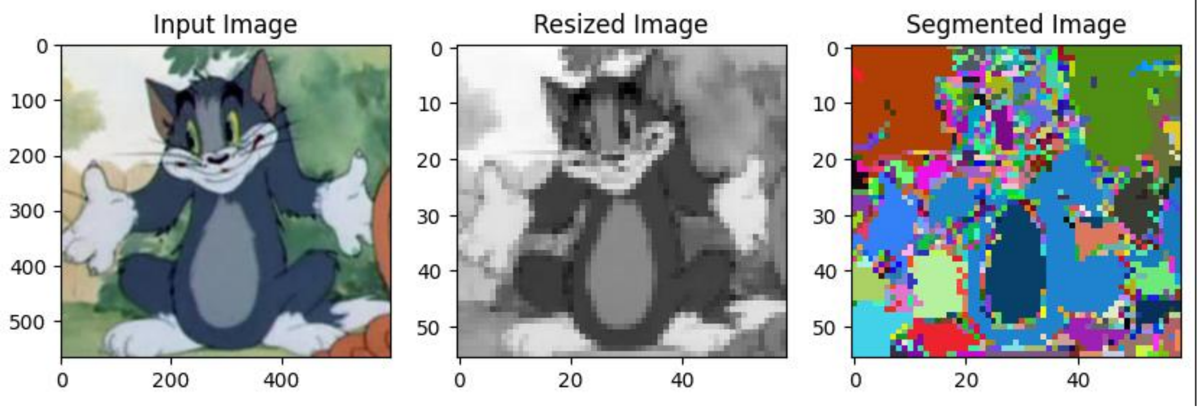


Figure 29: Original input image, resized greyscale image and segmented image with edge weight limit as 5 with image resolution of 56*59 pixels

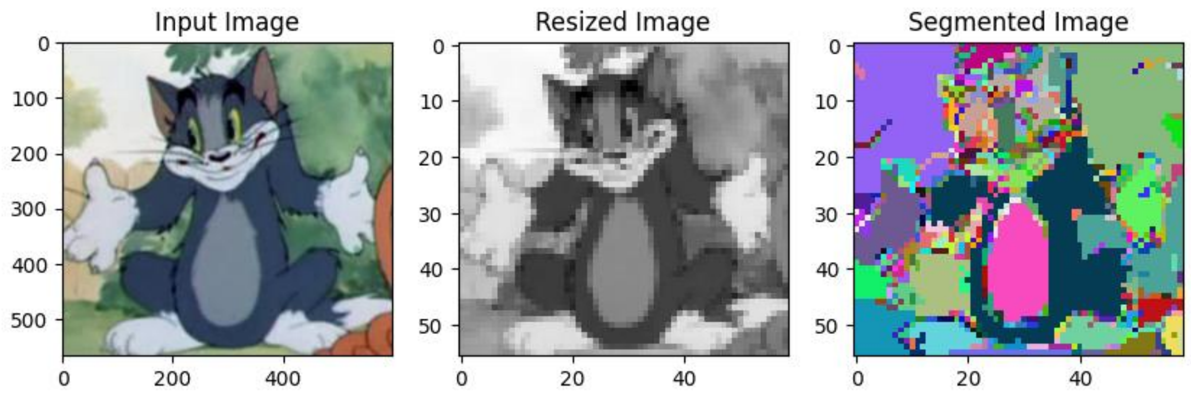


Figure 30: Original input image, resized greyscale image and segmented image with edge weight limit as 7 with image resolution of 56*59 pixels

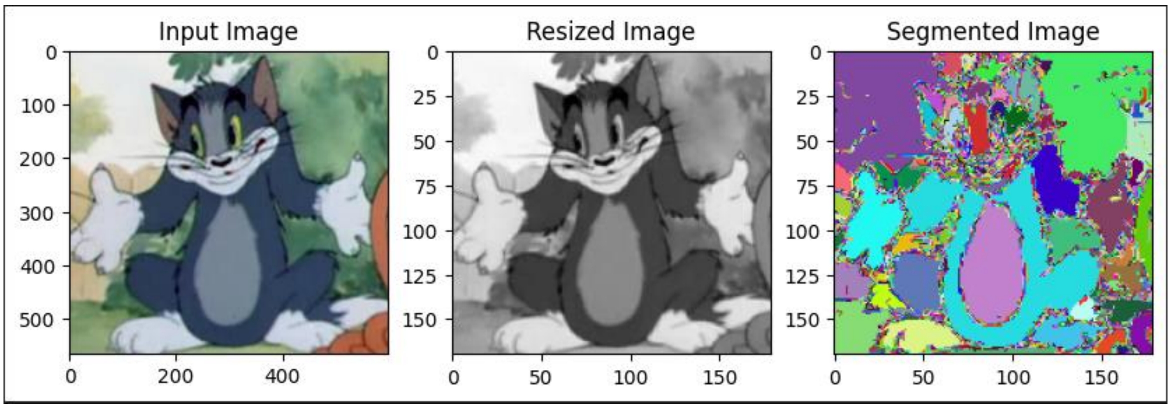


Figure 31: Original input image, resized greyscale image and segmented image with edge weight limit as 3 with image resolution of 170*179 pixels

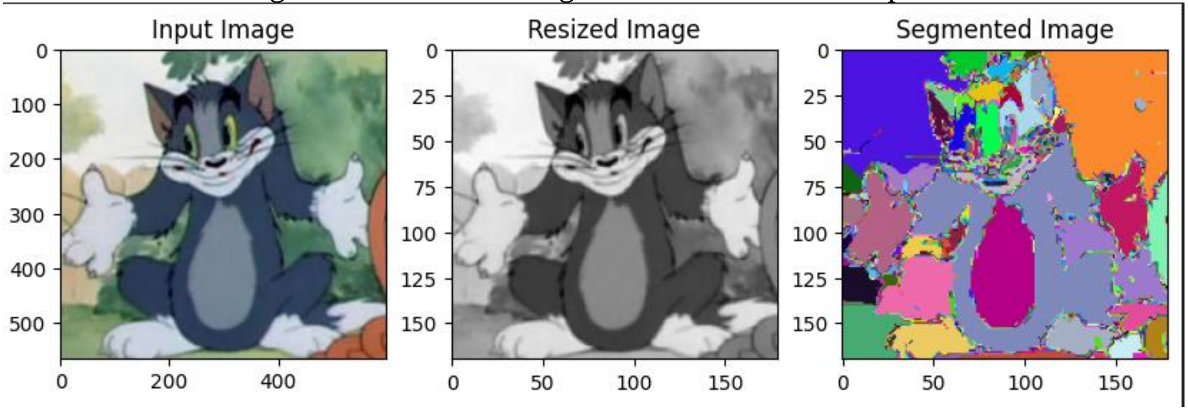


Figure 32: Original input image, resized greyscale image and segmented image with edge weight limit as 5 with image resolution of 170*179 pixels

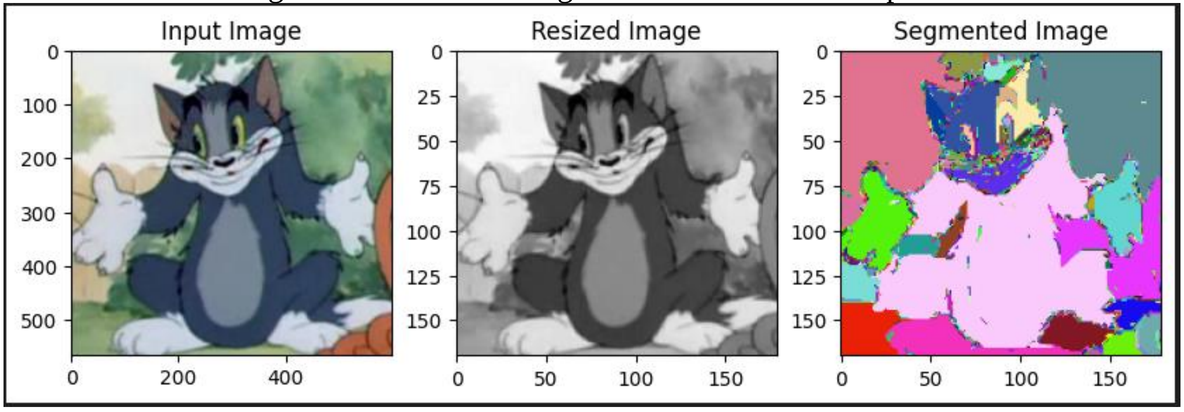


Figure 33: Original input image, resized greyscale image and segmented image with edge weight limit as 7 with image resolution of 170*179 pixels

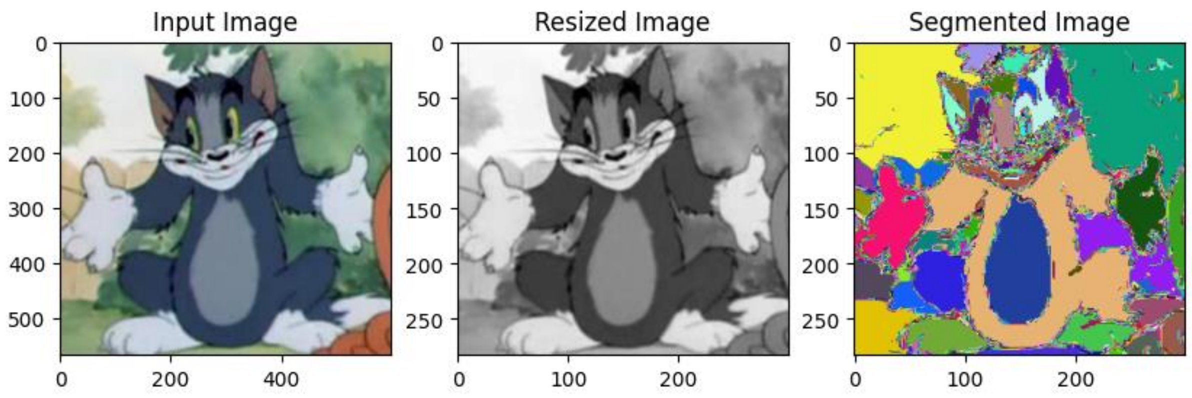


Figure 34: Original input image, resized greyscale image and segmented image with edge weight limit as 3 with image resolution of 283*299 pixels

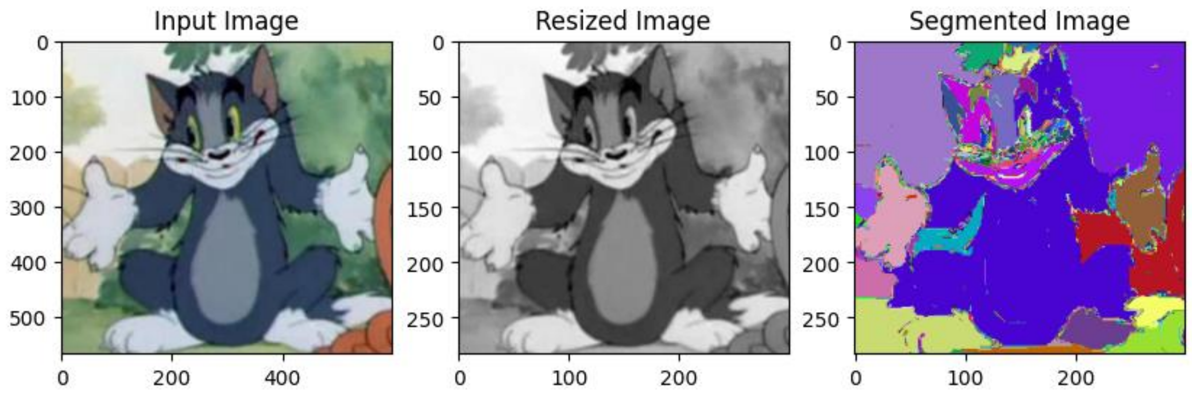


Figure 35: Original input image, resized greyscale image and segmented image with edge weight limit as 5 with image resolution of 283*299 pixels

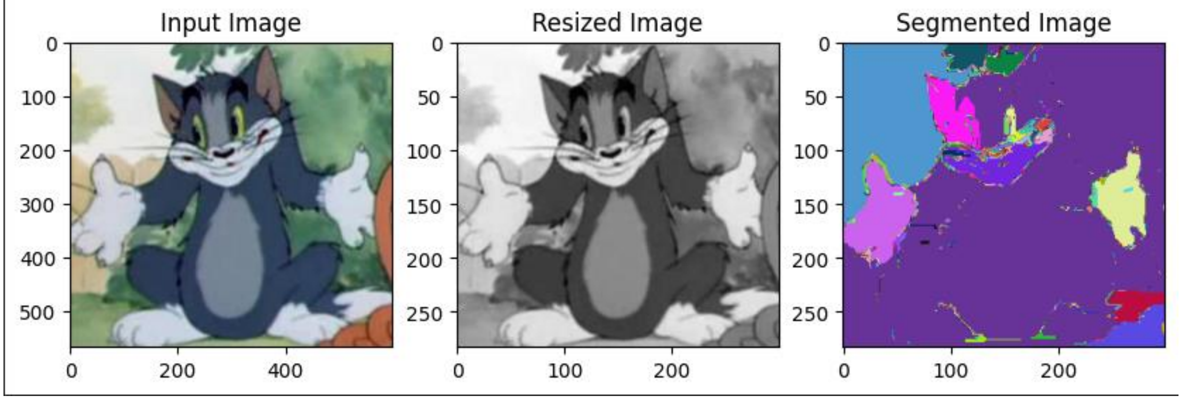


Figure 36: Original input image, resized greyscale image and segmented image with edge weight limit as 7 with image resolution of 283*299 pixels

From the image result we can observe that, the value of image resolution is inversely proportional to the weight limit.

Size 1: In the experiment cases, size 1 is the smallest image resolution among the other two. From Figure 28 – Figure 30 can we get a conclusion that the image is clearer to read in a higher weight limit variable comparing with the two higher options.

Size 2: For the middle case, from Figure 31 – Figure 33 we can see that the situation is obviously changing. For the middle level of weight limited, it is the clearest image to read. However, for the highest level of weight limit, the regions should not be merged are accepted by the algorithm to combine two trees as one.

Size 3: For the case in highest resolution, the situation of combining regions is more serious. From Figure 34 – Figure 36 we can see that, the object and the background are merged and cannot read the object clearly.

There may be multiple reasons for this appearance. One of the reasons to explain this appearance is that while the image resize process, when OpenCV resizes an image input into a small size, it will try to reduce the number of pixels while maintaining the pixels' colour or intensity level to preserve as much of the original image's detail and quality as possible. In this case, the pixel colour or intensity level difference between neighbouring pixels may become more extensive. Therefore, for images in smaller resolution (size 3), the colour regions may be merged, and it is hard to read the objects, even with a high weight limit value. Besides, for images with much smaller resolution (size 1), the colour regions may be challenging to merge and considered single components even with a high weight limit value.

4. Conclusion

In this study, we are going to verify the efficiency of Borůvka's algorithm by comparing the result with Kruskal's algorithm and Prim's algorithm; we estimated that Borůvka's algorithm may have shorting processing time to find the minimum spanning trees from the graphs according to the time complexity comparing to Kruskal's algorithm and Prim's algorithm. However, in this study, we experimented and found that the processing time for Borůvka's algorithm may take longer than expected from an adjacency matrix input converted from an image by calculating the pixel intensity levels of adjacent pixels as the weight of edge incident to the adjacent pixels. The reason is that in each iteration, Borůvka's algorithm may be able to reduce a considerable number of components. For images in large resolution, it also takes much longer to check each edge for each pixel in the early stage of the process. As a result, the processing time may not satisfy the estimation.

Next, we obtained and applied the minimum spanning tree list to the image segmentation process and verified the relationship between the image resolution and weight limit value to consider if the pixels are connected. The result is that they are inversely proportional to each other: For larger image resolution, should apply a lower value of weight limit to prevent over-merged; For smaller image resolution, should apply a higher value of weight limit to adapt the higher difference between the pixels after the image resize processing.

Appendices

Appendix I	Check the functions used in this project from the below link: https://github.com/Ryniii/finalYearProject.git

References:

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.

[2] Felzenszwalb, P. F., & Huttenlocher, D. P. (2004). Efficient graph-based image segmentation. International Journal of Computer Vision, 59(2), 167-181.

[3] Graham, R. L., & Hell, P. (1985). On the history of the minimum spanning tree problem. IEEE Annals of the History of Computing, 7(1), 43-57.

[4] Python Software Foundation. (n.d.). heapq — Heap queue algorithm. Retrieved [your access date here, e.g., April 12, 2024], from https://docs.python.org/3/library/heapq.html

[5] Nesetril, J., Milkova, E., & Nesetrilova, H. (2001). Otakar Borůvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. Discrete Mathematics, 233(1-3), 3-36.

[6] NumPy. (n.d.). NumPy Introduction. Retrieved [your access date here, e.g., April 12, 2024], from https://numpy.org/doc/stable/user/absolute_beginners.html

[7] OpenCV. (n.d.). Introduction to OpenCV. Retrieved [your access date here, e.g., April 12, 2024], from https://opencv.org/about/

[8] Pal, N. R., & Pal, S. K. (1993). A review on image segmentation techniques. Pattern Recognition, 26(9), 1277-1294.

[9] Wang, X. F., & Chen, G. (2003). Complex network: small-world, scale-free and beyond. IEEE Circuits and Systems Magazine, 3(1), 6-20. doi: 10.1109/MCAS.2003.1228503.