Report for HEP503 Final Assessment Unit 2

Alan Kowalczyk

25th February 2024

Introduction

This report was created using LaTeX [7], a software for document preparation. All text has been written entirely by myself, with later grammatical corrections using Grammarly [1]. The program has been written using Python [3] version 3.10.12 in Visual Studio Code [4] version 1.85.2. All program tests were carried out on a laptop equipped with a 16-core i9 CPU, clocked at 5.3 GHz, and 64GB of RAM, operating under Ubuntu 22.04.3 LTS. Class diagrams have been created using GitUML [2]. The program's primary goals are create a SHA256 checksum, instantiate a series of objects, generate a graph, compute an MST and render an MST as an image. All tasks have been described in this report.

1 Structure of the program

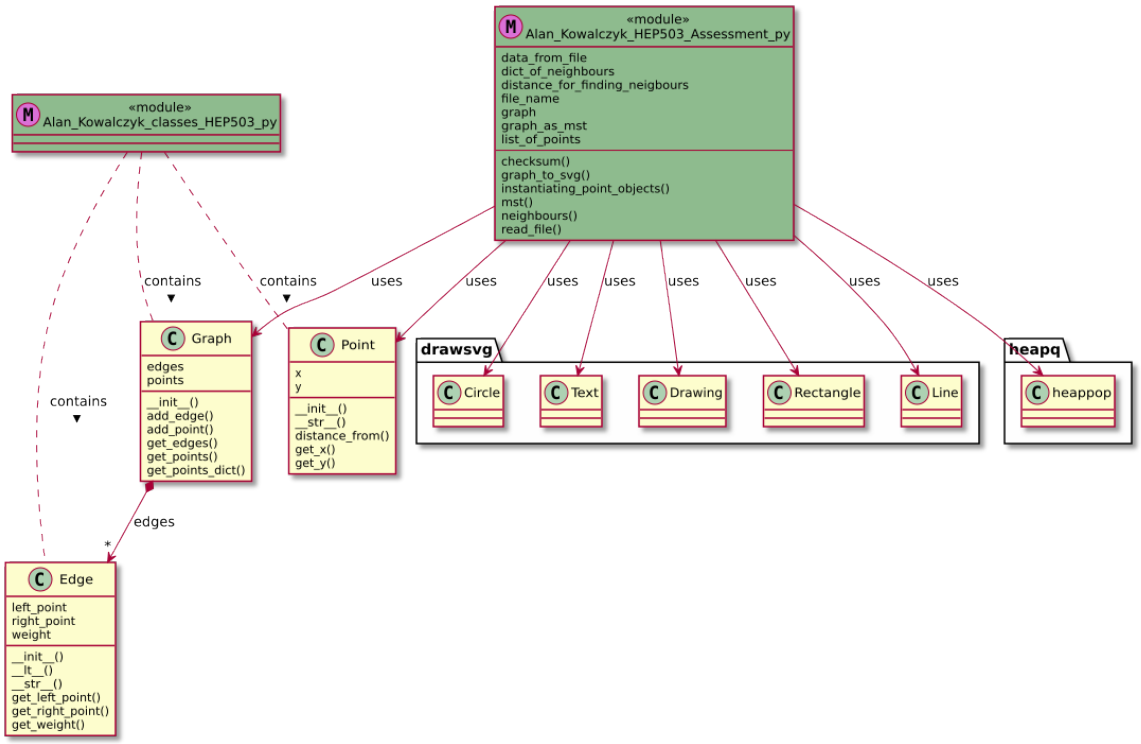

Figure 1: Structure of the program

The program has been modularised for readability, and its structure is demonstrated in Figure 1. One file contains only classes, while the second contains all function declarations and demos. In both files in the docstrings, I included information about the author, date of creation and date of last modification. As per the main file, I decided to keep the variables **file_name** and **distance_for_finding_neighbours** at the beginning of a script, as this can help easily change the file name or quickly change distance within we are looking for neighbours. Next, I included all **import** statements, followed by all **function** definitions.

1.1 Reading data from a file

```
def read_file(file_name):
    """
    read_file - a function which is responsible for reading data from a
                                            file
    input:
        file_name - - a string - name of file with data
    output:
        data - a list - list of y values as strings
    """
    try:
        with open(file_name, 'r') as f:
            for row in f:
                data = row.strip().split(",")
        return data
    except FileNotFoundError:
        print(f"There is no such file: {file_name}, check directory and
                                                spelling.")
```

A **read_file** function is responsible for reading data from a file and exception handling if a file is unavailable. I decided to handle exceptions using **Try...Except** blocks, which execute a block of code in a **try** block, except when the file is not found, which is handled by an **except** block of code. I used a built-in **open()** function to open a file. Statement **with**, is used because there is no need to close a file, as **with** statement automatically closes the file after use. Data read from the file are read as a **string**, so I used built-in **string** methods **.strip()** - which allowed me to get rid of white spaces (' ', '\t', '\n') and **.split(',')** which allow me to split data using **','** as the delimiter. This function returns a Python **list** containing data from the file as a **string**.

1.2 Checking the validity of test data

A **checksum()** function accepts a Python **list** containing string values as input, next using a **for** loop function to traverse through the **list**, concatenating all values into one string. I used the **sha256()** function imported from the **Hashlib** module for hashing. Using a **.encode('utf-8')** method was necessary to properly encode data using 'utf-8'. A **hexdigest()** method returns a checksum in hexadecimal digits. The function prints concatenated y values from input, expected and calculated hash and returns Boolean value: **True** if the calculated checksum is identical with the expected value, or **False** otherwise. [1]

[1] Output of validity of test data in Appendix B

```
def checksum(data):
    """
    checksum - a function which checks the validity of data using
                                        SHA256 checksum
    input:
        data - a list - list of y values as strings
    output:
        - a boolean - True if the checksum matches the provided
                                            checksum in the helper file
                                            , False otherwise
    """
    data_to_hash = ''
    for y in data:
        data_to_hash += y
    expected_checksum =
    "5c14e4599f1d2a39abe6b487ac2a5415c894c6882f5fdd4a40e02c7dd628829a"
    hashed_data = sha256(data_to_hash.encode('utf-8')).hexdigest()
    print(f"Concatenated y values from the file presented below: \n{
                                        data_to_hash}")
    print(f"Validity of test data by calculating SHA256 checksum: \n
                                        Actual checksum: {hashed_data}
                                        \nExpected checksum: {
                                        expected_checksum}")
    return(f"Is a match?:       {hashed_data == expected_checksum}")
```

1.3 Implementing Point class and instantiating Point objects

```
def instantiating_point_objects(data):
    """
    instantiating_point_object - a function which is instantiating
                                        Point objects
    input:
        data - a list - a list of y value as str
    output:
        points_list - a list - a list of Points objects
    """
    points_list = []
    x = 1
    for y in data:
            point = Point(x, int(y))
            points_list.append(point)
            x += 1
    return points_list
```

Implementation of the Point class is described in subsection 1.5.1. An **instantiating_point_objects()** function accepts a **list** of string values as input. Inside a **for** loop function is creating an object of class **Point** using as **x** values numbers from 1 to a number of elements in the **list**. For corresponding **y** values, values from the input list are transformed to integer values using the built-in **int()** function. Next, the Point object is appended to a list of Point objects. The function returns a Python **list** containing objects of class **Point**.

1.4 Finding neighbours

```
def neighbours(origin, list_of_points, distance):
    """
    neighbours - a function which is finding all neighbours from origin
                                        within a {distance} range
    input:
        origin - an object - a Point object
        list_of_points - a list - a list for looking for neighbours
        distance - an int - distance for finding neighbours
    output:
        neighbours_list - a list - a list of neighbours
    """
    neighbours_list = []
    for point in list_of_points:
        dist = origin.get_distance_from(point)
        if dist <= distance and dist > 0:
            neighbours_list.append(point)
    return neighbours_list
```

Function **neighbours** accept as input an object of class **Point** and an integer value - a distance within a function is looking for neighbours of our Point object. The function finds all neighbours in the distance (0, **distance**] using a **.distance_from()** method (described in subsection 1.5.1) and returns a list of all neighbours within **distance**.

1.5 Implementing of Point, Graph and Edge Classes

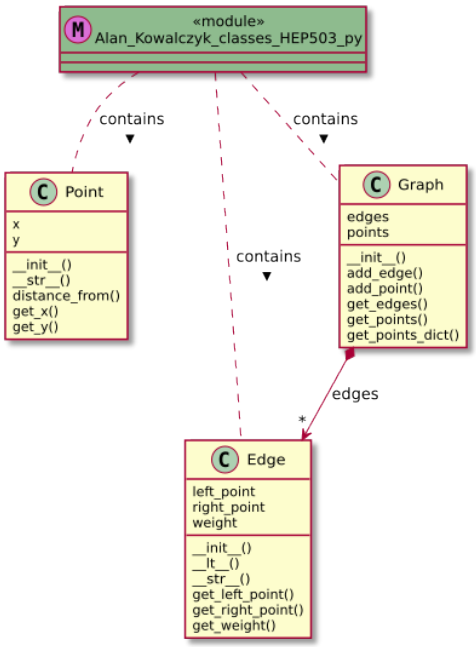

Figure 2: Structure of classes

Figure 2 represents a simple class diagram of classes used in this program, while the source code of implementation is provided in Appendix A.

1.5.1 Class Point

The constructor of the Point class takes two arguments, **x** and **y**. These arguments are stored in the **x** and **y** instance attributes, respectively, as Cartesian coordinates. A **.__str__()** method returns a string representation of an instance of objects class Point in representation <x,y>. The Function has no setters and contains three getters, **.get_x()** and **.get_y()**, return arguments **x** and **y** as integers. Meanwhile, **.get_distance_from(other)** takes as argument other instance of class Point and returns Euclidean distance between self and other as a float, calculated as:

$$distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{1}$$

where:

- distance - is calculated distance between both instances of objects class Point
- x_1, y_1 - Cartesian coordinates of one instance of class Point
- x_2, y_2 - Cartesian coordinates of the second instance of class Point

1.5.2 Class Edge

The object of class Edge can not exist outside of class Graph. Moreover, there have to be at least two objects of class Point to object Edge exist. As per graph theory, an Edge [2] is a line joining two Points [3]. The constructor of Edge takes two arguments, **left_point** and **right_point**, which are instances of class Point, and calculates the weight of edge using a **.get_distance_from(other)** method of class Point. A **.__str__()** method returns a string representation of instances of objects class Edge. A **.__lt__()** method accepts one argument, which is another instance of class Edge, and returns the Boolean value "True" if the weight of **self** is less than the weight of **other**, "False" otherwise. Function doesn't have any setters and contains three getters **.get_left_point()**, **.get_right_point()** returns arguments **left_point** and **right_point** as objects of class Point. Meanwhile, **.get_weight()** returns the weight of the edge as a float value.

1.5.3 Class Graph

The constructor of class Graph doesn't take any argument and initialise an instance of class Graph as an empty object, with two attributes **points** as a dictionary for keeping as a key Point object, and as a value a list of neighbours of this point, and **edges** which is a list containing all edges in a graph. There are two setters: **.add_point (point)** and **.add_edge (edge)**. The first setter takes as input an instance of class Point and adds this to an object of class Graph as Point [4]. The second setter takes as an input instance of class Edge and adds it to the graph. It's worth noticing that the **.add_edge (edge)** method instantiates two objects of class Point and adds them to a Graph if those weren't previously explicitly declared, as an edge must exist between two points and can not exist without them. It is possible to have a graph object with one point

[2] Edge is also called Link or Arc[6]

[3] Point is also called Vertex or Node

[4] name Vertex or Node is more common; however, I decided to use the name Point to be consistent with the previous implementation of class Point - an instance of class Point can exist without Graph, while Vertex is part of the graph

and no edges, but it's impossible to get an edge without points. Class Graph contains three getters **.get_points()**, **.get_edges()** returns a list of objects class Point and a list of objects class Edge, respectively. Meanwhile, **.get_points_dict()** returns a dictionary mapping each object of class Point to its neighbours.

1.6 Generating the Minimum Spanning Tree

```
def mst(graph):
    """
    mst - a function, which is generating a Minimum Spanning Tree
    input:
        graph - an object - an object class Graph
    output:
        mst - an object - an object class Graph as MST
    """
    sorted_edges = sorted(graph.get_edges(), key = attrgetter("weight")
                                                  )
    dict_of_points = graph.get_points_dict()
    mst = Graph()
    # Adding the first shortest edge and both points to the MST
    mst.add_edge(sorted_edges[0])
    # Creating a priority queue
    priority_queue = []

    # Add all edges between existing points in MST and their neighbours
                                           to the priority queue
    for point in mst.get_points():
        for neighbour in dict_of_points[point]:
            if neighbour not in mst.get_points():
                heapq.heappush(priority_queue, Edge(point, neighbour))

    while len(mst.get_points()) < len(graph.get_points()):
        # Get the edge with the smallest weight from the priority queue
        edge = heapq.heappop(priority_queue)
        right_point = edge.get_right_point()

        if edge not in mst.get_edges() and right_point not in mst.
                                                get_points():
            mst.add_edge(edge)
            # Add edges between the new point and its neighbours to the
                                                    priority queue
            for neighbour in dict_of_points[right_point]:
                if neighbour not in mst.get_points():
                    heapq.heappush(priority_queue, Edge(right_point,
                                                        neighbour))
    return mst
```

The **mst()** function takes an instance of a graph object, as input. I used Prim's algorithm to calculate an MST, as there will be one object at each stage with all points connected by edges together, which will grow in each step. Firstly, the function gets a list of edges using the **.get_edges()** method and sorts them using the built-in **sorted()** function. Using an **attrgetter()** function from the **operator** module, I specify that sorting should be based on the **"weight"** attribute of an instance of class Edge in order to compare two instances of the class Edge method **.__lt__()** is used. The function creates an empty object class Graph and adds the first shortest edge from the sorted list of edges

using the **.add_edge()** method of class Graph; there is one edge and two points in the graph at this moment, as this method is adding points between which edge exist as well. The next step is adding all edges between points already in the MST and all of its neighbours, which have not been added to the MST yet, to a priority queue. I decided to use a priority queue implemented using Heaps, a better option for creating priority queues than linked list (Agarwal, 2022, chapter 7 [5]). Function **mst()** uses **.heappush()** and **.heappop()** functions from the **heapq** module to push and pop items to/from the priority queue, respectively. As a next step, until all objects class Point from the input graph are added to the MST, the function takes the shortest edge from the priority queue, adds this edge to the MST using the **.add_edge()** method, and adds all edges between added point and all of its neighbours to the priority queue.

1.7 Rendering the MST as an image

```
def graph_to_svg(graph, filename):
    """
    graph_to_svg - a function, which is drawing an object class Graph()
                                            as an SVG image
    source HEP503 from Abertay University, unit 7.5, accessed: 21 Feb
                                           2024, modified
    input:
        graph - an object - an object type Graph for drawing
        filename - a string - filename to which function will save a
                                               file
    output:
        none
    """
    height = 800
    width = 800
    padding = 20 #to keep points away from the edge of the drawing
    today_date = dt.date.today()
    d = drawsvg.Drawing(width, height)
    # Add a white rectangle to be the background of the drawing
    d.append(drawsvg.Rectangle(0, 0, width, height, fill = "white"))
    for edge in graph.get_edges():
        # Flip the y-coordinates by subtracting from height, so the
        # start point of axes (0,0) will be bottom left
        d.append(drawsvg.Line((edge.get_left_point().get_x() * 7.5 +
            padding), (height - edge.get_left_point().get_y() * 7.5 -
            padding), (edge.get_right_point().get_x() * 7.5 + padding),
            (height - edge.get_right_point().get_y() * 7.5 - padding),
            stroke = "blue", stroke_width = 2, fill = "none"))
    for point in graph.get_points():
# Adjust positions so the start point of axes (0,0) will be bottom left
        d.append(drawsvg.Circle((point.get_x() * 7.5 + padding),
            (height - point.get_y() * 7.5 - padding), 5, fill = "red",
            stroke_width = 2, stroke = "black"))
    d.append(drawsvg.Text(f"author: Alan Kowalczyk, created on {
                                           today_date}", 14, 275, 15))
    with open(f"{filename}.svg", 'w') as f:
        f.write(d.as_svg())
```

The **mst()** function takes an instance of a graph object as input. I used **Line()** and **Circle()** functions to represent edges and points. In addition, I used a **Text()** function to add author's name and date of creation to an SVG image. The creation date is obtained using a **datetime** module and its function **today()** from the submodule **date**. All functions for drawing are from **drawsvg** module. Results of using a **graph_to_svg()** function are represented in Figures 3 and 4 for rendering a whole graph object with all edges and the MST, respectively.

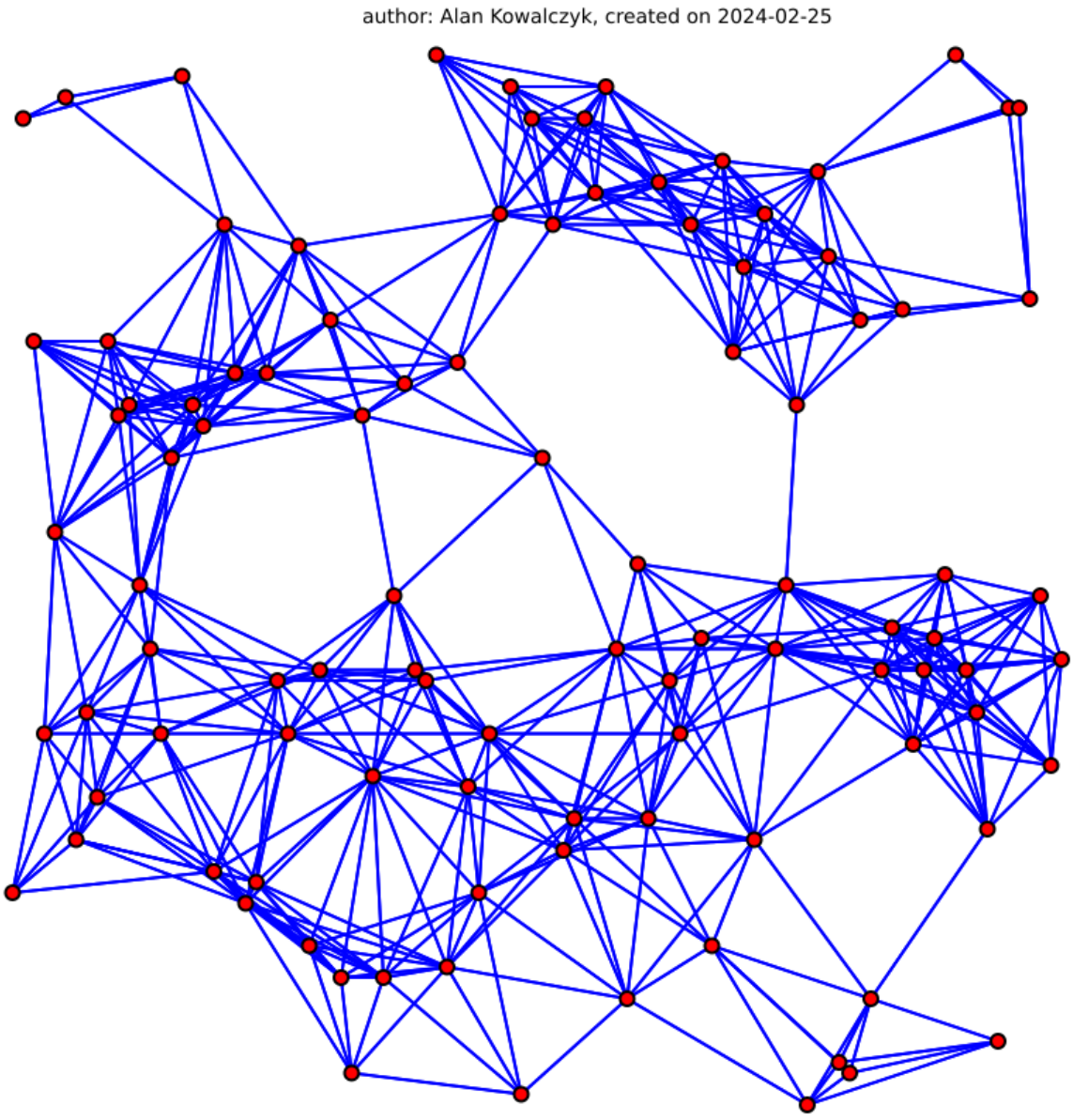

Figure 3: Rendered graph as an image

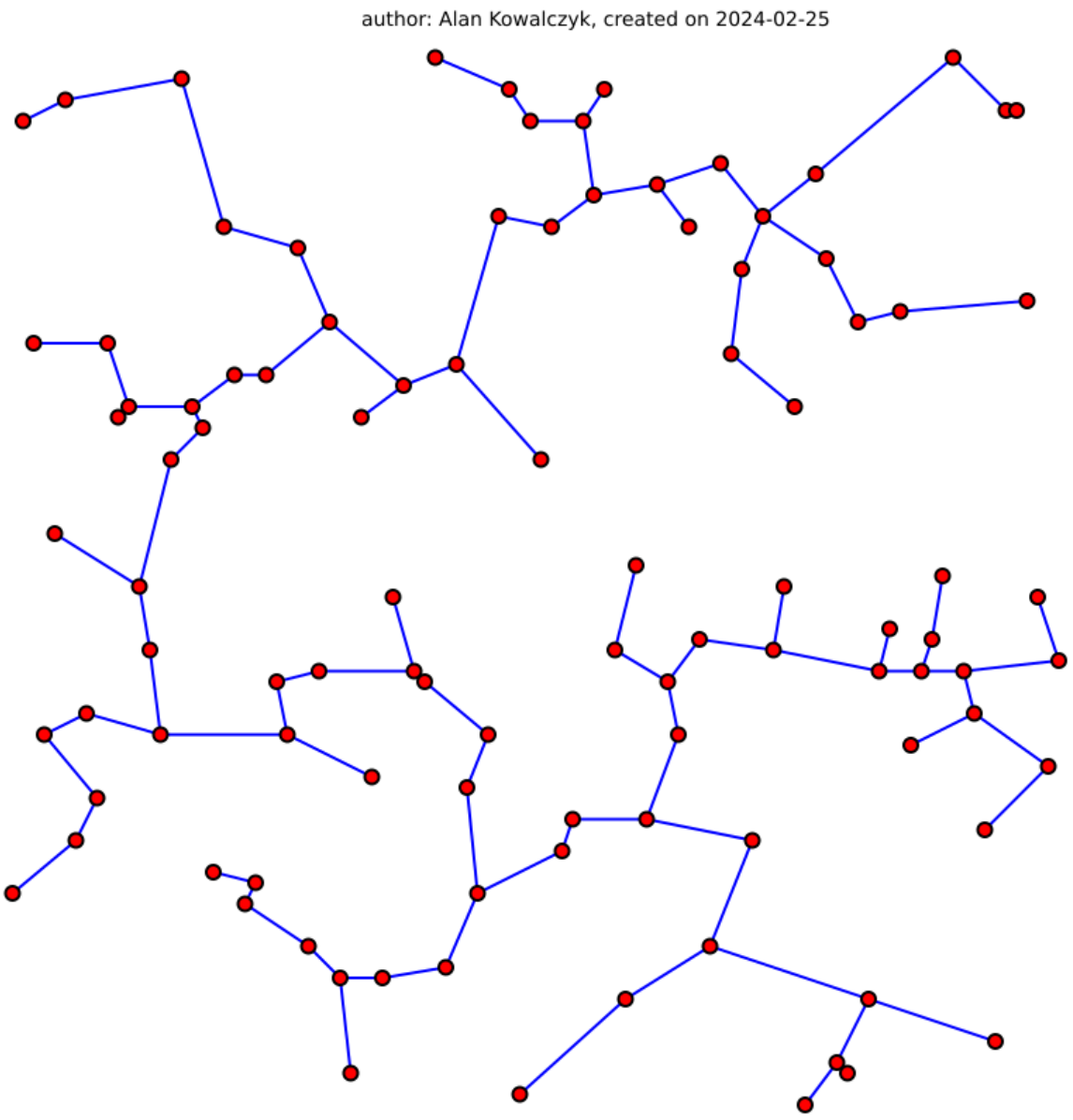

Figure 4: Rendered MST as an image

References

[1] Grammarly - typing assistant. `https://app.grammarly.com/`. Online, last accessed 25 February 2024.

[2] Official gituml webpage. `https://www.gituml.com/`. Online; last accessed 24 February 2024.

[3] Official python documentation. `https://www.python.org/`. Online; last accessed 01 February 2024.

[4] Official visual studio code webpage. `https://code.visualstudio.com/`. Online; last accessed 01 February 2024.

[5] Basant Agarwal. *Hands-On Data Structures and Algorithms with Python (Third Edition)*. Packt Publishing, 2022.

[6] John V. Guttag. *Introduction to Computation and Programming Using Python With Application to Computational Modeling and Understanding Data (Third Edition)*. The MIT Press, 2021.

[7] LaTeX Project Team. Latex - a high-quality typesetting system. `https://www.latex-project.org/`. Online, last accessed 18 December 2023.

A Appendix A: Source code of Classes

```
"""
Created on 23 Feb 2024
Last modification 25 Feb 2024 by AK

author Alan Kowalczyk
"""


class Point(object):
    """
    a class of Point()
    """

    def __init__(self, x, y):
        """
        the constructor of a class, initialise with coordinate <x,y>
        """
        self.x = x
        self.y = y

    def __str__(self):
        """
        return a string representation of coordinates in format <x,y>
        """
        return "<" + str(self.x) + "," + str(self.y) + ">"

    def get_distance_from(self, other):
        """
        return a float distance between {self} object and {other}
                                            object
        """
        x_dist = self.x - other.get_x()
        y_dist = self.y - other.get_y()
        return (x_dist ** 2 + y_dist ** 2) ** 0.5

    def get_x(self):
        """
        return a x value as int
        """
        return self.x

    def get_y(self):
        """
        return a y value as int
        """
        return self.y
```

```
class Edge(object):
    """
    class Edge() connecting two objects of class Point()
    """
    def __init__(self, left_point, right_point):
        """
        the constructor of a class, initialise with two Point() objects
                                            between which Edge() is
                                           connecting
        """
        self.left_point = left_point
        self.right_point = right_point
        self.weight = left_point.get_distance_from(right_point)

    def __str__(self):
        """
        return a string representation of the Edge()
        """
        return str(self.left_point) + '<->' + str(self.right_point)

    def get_left_point(self):
        """
        return a left point of class Point()
        """
        return self.left_point

    def get_right_point(self):
        """
        return a right point of class Point()
        """
        return self.right_point

    def get_weight(self):
        """
        return a weight of the {edge}, which is the distance between {
                                           points}, as float
        """
        return self.weight

    def __lt__(self, other):
        """
        return True if weight of {self} is smaller than weight of {
                                           other}, False otherwise ->
                                           for comparison in Priority
                                           Queue (hepq)
        """
        return self.weight < other.weight
```

```
class Graph(object):
    """
    class Graph() containing objects of class Point(), connected by
                                        object(s) class Edge()
    """
    def __init__(self):
        """
        the constructor of a class, initialise empty
        """
        self.points = {}
        self.edges = []

    def add_point(self, point):
        """
        adding an {point} object to a Graph()
        """
        if point not in self.points:
            self.points[point] = []

    def add_edge(self, edge):
        """
        adding an {edge} between {points}, and if {points} are not
                                        previously explicitly
                                        declared, then add {points}
                                         too, as the {edge} has to
                                        connect two {points{} and
                                        can not exist without {
                                        points}
        """
        if edge.get_left_point() not in self.points:
            self.add_point(edge.get_left_point())
        if edge.get_right_point() not in self.points:
            self.add_point(edge.get_right_point())
        self.points[edge.get_left_point()].append(edge.get_right_point
                                        ())
        self.edges.append(edge)

    def get_points(self):
        """
        return a list of {points}
        """
        return list(self.points.keys())

    def get_edges(self):
        """
        return a list of {edges}
        """
        return self.edges

    def get_points_dict(self):
        """
        return a dictionary mapping each {point} to their neighbours
        """
        return self.points
```

B Appendix B: Output of running program

```
36
37
38  def checksum(data):
39      """
40      checksum - a function which checks the validity of data using SHA256 checksum
41      input:
42          data - a list - list of y values as strings
43      output:
44          - a boolean - True if the checksum matches the provided checksum in the helper file, False otherwise
45      """
46      data_to_hash = ''
47      for y in data:
48          data_to_hash += y
49      expected_checksum = "5c14e4599f1d2a39abe6b487ac2a5415c894c6882f5fdd4a40e02c7dd628829a"
50      hashed_data = sha256(data_to_hash.encode('utf-8')).hexdigest()
51      print(f"Concatenated y values from the file presented below: \n{data_to_hash}")
52      print(f"Validity of test data by calculating SHA256 checksum: \n  Actual checksum: {hashed_data} \nExpected checksum: {expected_checksum}")
53      return(f"Is a match?:       {hashed_data == expected_checksum}")
54
```

```
PROBLEMS 828  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

alan@alan-64gb:~/Abertay/HEP503/Alan_Kowalczyk_Assessment_Unit2$ /bin/python3 /home/alan/Abertay/HEP503/Alan_Kowalczyk_Assessment_Unit2/Alan_Kowalczyk_HEP503_
Assessment.py
Concatenated y values from the file presented below:
209372355495253729726566494335619766642283691921694035811541741236531124868414099137030203584961936183242793869643105127874035834415897179258443496608880437410414575344144509941372669494764832 42
Validity of test data by calculating SHA256 checksum:
  Actual checksum: 5c14e4599f1d2a39abe6b487ac2a5415c894c6882f5fdd4a40e02c7dd628829a
Expected checksum: 5c14e4599f1d2a39abe6b487ac2a5415c894c6882f5fdd4a40e02c7dd628829a
Is a match?:       True
alan@alan-64gb:~/Abertay/HEP503/Alan_Kowalczyk_Assessment_Unit2$
```