Assignment 3 - CSCI203 - 7161955

Solution Understanding

High-Level Description

Pseudo Code Based on Lectures

```
start script
Capture user input
Initialise variables
read in file input
repeat
  for input_line in file_vertexs
    vertexList.append(Vertex(input_line))
  rof

  for input_line in file_edge
    edgeList.append(Edge(input_line))
  rof

  repeat
    currentIndex = minHeap.pop()
    if currentIndex == endVertex
      Return endVertex.distance, path
    fi

    for edge in edges
      if edge.start == currentIndex.label:
        neighbour = next(vertexList)
        if current.distance + edge.weight < neighbour.distance
          unvisitedPoints.push(neighbour)
        fi
      fi
    rof

  until minHeap is not empty

  #dfs function
  if currentVertex == goal_vertex
    for i in range(path)
      Path[i] = current_path[i]
    rof
  else
    For edge in edges:
      if edge.start == current_vertex.label and neighbour not in visited:
        dfs_longest_path(vertices, edges, neighbor, goal_vertex, visited, current_weight +
edge.weight, max_weight, path, current_path, path_index, max_path_index)
      fi
```

```
    rof

print Graph Statistics
finish
```

Pseudo Code in English

```
initialise
set variables
Take user input for filename
read in input file
  Add input to vertex_array
  Add input to edge_array

while unvistedPoints isn't empty:
  currentVertex = unvistedPoints.pop()
  If currentVertex == endVertex
    return endVertex.distance, path

  for edge in edges
    Neighbour = next_vertex
    If edge.start == currentVertex.label and (current_vertex.distance + end.weight <
neighbour.distance)
      unvistedPoints.push(neighbour)

dfs(vertices, edges, current_vertex, goal_vertex, visited, current_weight, max_weight,
path, current_path, path_index, max_path_index)
  if current_vertex == goalVertex
    for i in range(path_index)
      Path[i] = current_path[i]
  else
    for edge in edges
      if edge.start == current_vertex.label
        If neighbour is not in visited
          dfs(vertices, edges, neighbor, goal_vertex, visited, current_weight + edge.weight,
max_weight, path, current_path, path_index, max_path_index)

print statistics to terminal
```

Complexity Analysis of Solution

My solution makes use of two different algorithms one for finding the shortest path and the other for finding the longest path.

Depth First Search Algorithm

The complexity of DFS is $O(V + E)$ that DFS traverses every Vertex and every Edge at most one time. This means the worst case is the addition of the length of the two inputs (Vertex and Edge).

Dijkstra Algorithm

The complexity of this algorithm $0((V + E) \cdot logV)$.

- $V + E$: Total number of iterations we go through for the algorithm
- $logV$: This is the time complexity associated with minHeap operations such as Insert, Delete and Key Update.
- $0((V + E) \cdot logV)$: The total number of iterations multiplied by the associated complexity of the underlying data structure and its operations.

Total Complexity

To find the complexity of the solution we combine the complexities of the worst-case scenarios of the Dijkstras Algorithm and Depth Search First which gives us $0((V + E) \cdot logV) + 0(V + E)$.

Data Structures Used

Custom Classes

Vertex

The vertex class represents the vertex in the weighted graph, with attributes for the label (String), coordinates (float, float), and distance (float) between other points. I implemented this myself, including some additional class functions for string representation and for distance comparisons.

```python
class Vertex:
    def __init__(self, label, x, y):
        """
        Initializes a Vertex object.
        Args:
            label: The label of the vertex (usually an integer or string).
            x: The x-coordinate of the vertex.
            y: The y-coordinate of the vertex.
        """
        self.label = label
        self.x = x
        self.y = y
        self.distance = float('inf')  # Distance from the source vertex, used in shortest path algorithms

    def __lt__(self, other):
        """
        Less-than comparison based on distance.
        Args:
            other: Another Vertex object to compare with.
        Returns:
            True if this vertex's distance is less than the other vertex's distance.
        """
        return self.distance < other.distance

    def __str__(self):
        """
        String representation of the vertex.
        Returns:
            The label of the vertex as a string.
        """
        return str(self.label)

    def __repr__(self):
        """
        Official string representation of the vertex.
        Returns:
            The label of the vertex as a string.
        """
        return self.__str__()
```

Edge

The Edge class represents an edge in a weighted graph with attributes for the start vertex label (String), end vertex label (String), and the weight (float) of the edge. This class was implemented to encapsulate the properties of an edge in the graph, including the vertices it connects and the cost associated with traversing it. Additionally, the class includes methods for string representation.

```
class Edge:
    def __init__(self, start, end, weight):
        """
        Initializes an Edge object.
        Args:
            start: The label of the start vertex.
            end: The label of the end vertex.
            weight: The weight of the edge.
        """
        self.start = start
        self.end = end
        self.weight = weight

    def __str__(self):
        """
        String representation of the edge.
        Returns:
            A string in the format "start end weight".
        """
        return str(self.start) + " " + str(self.end) + " " + str(self.weight)

    def __repr__(self):
        """
        Official string representation of the edge.
        Returns:
            A string in the format "start end weight".
        """
        return self.__str__()
```

MinHeap

The MinHeap class represents a minimum heap data structure, which is a complete binary tree where the value of each node is less than or equal to the values of its children. This class was implemented to support priority queue operations, such as inserting elements and extracting the minimum element. The class includes methods for adding elements (push), removing the minimum element (pop), and checking if the heap is empty (empty).

```python
class MinHeap:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def leftChild(self, i):
        return 2 * i + 1

    def rightChild(self, i):
        return 2 * i + 2

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def heapifyUp(self, i):
        while i > 0 and self.heap[self.parent(i)] > self.heap[i]:
            self.swap(i, self.parent(i))
            i = self.parent(i)

    def heapifyDown(self, i):
        largest = i
        left = self.leftChild(i)
        right = self.rightChild(i)

        if left < len(self.heap) and self.heap[left] < self.heap[largest]:
            largest = left

        if right < len(self.heap) and self.heap[right] < self.heap[largest]:
            largest = right

        if largest != i:
            self.swap(i, largest)
            self.heapifyDown(largest)

    def push(self, item):
        self.heap.append(item)
        self.heapifyUp(len(self.heap) - 1)

    def pop(self):
        if len(self.heap) == 0:
            return None

        root = self.heap[0]
        self.heap[0] = self.heap[-1]
        self.heap.pop()
        self.heapifyDown(0)
        return root

    def empty(self):
        return len(self.heap) == 0
```

Python Inbuilt Classes

The list class in Python represents a dynamic array that can hold an ordered collection of items. They support various operations such as indexing, slicing, appending, and extending. Lists can contain elements of different types, including other lists. For my use case, we just use it to append and remove elements instead of all of its additional niceties.

Terminal Output

Note: This will take some time to run it is not very fast