High-level plan

1. **Input Handling:**
   - Read the name of the input file from the console.
   - Parse the file to extract:
     - Number of vertices and edges.
     - Coordinates of each vertex.
     - The weighted directed edges between vertices.
     - The start and goal vertices.
2. **Graph Representation:**
   - Use an **adjacency list** to represent the graph. Each vertex will have a list of neighbouring vertices and associated edge weights.
   - Store vertex coordinates to calculate Euclidean distances as required.
3. **Shortest Path Algorithm:**
   - Use **Dijkstra's Algorithm** to find the shortest path between the start and goal vertices.
   - Track the path and total distance.
4. **Longest Path Algorithm:**
   - **Dynamic programming** can be used to find the longest path for acyclic graphs. Since the graph may contain cycles, a modified **Depth-First Search (DFS) algorithm** with backtracking is applied to find the longest simple path.
5. **Output:**
   - The number of vertices and edges.
   - The start and goal vertices.
   - The Euclidean distance between the start and goal vertices.
   - The vertices on the shortest path and its length.
   - The vertices on the longest path and its length.

Pseudocode

1. Parse the input file:

```
function read_file(filename):
  open file
  read n (number of vertices) and m (number of edges)
  initialize vertices as empty list
  for i in range(n):
    read vertex label, x, y coordinates
    store in vertices list
  end for

  initialize graph as list of empty lists with size n
  for j in range(m):
    read start_vertex, end_vertex, weight
    add (end_vertex, weight) to graph[start_vertex]
  end for

  read start_vertex, goal_vertex
  return graph, vertices, start_vertex, goal_vertex
```

Name: Michael McMillan
Student#: 8116775

2. Calculate the Euclidean distance:

```
function euclidean_distance(v1, v2):
  return sqrt((v1[0] - v2[0])^2 + (v1[1] - v2[1])^2)
```

3. Class: PriorityQueue

```
class PriorityQueue:
  function __init__(size):
    create elements array of size (size, 2) initialised to zeros
    set capacity to size
    set size to 0

  function is_empty():
    return size == 0

  function push(priority, item):
    if size >= capacity:
      raise RuntimeError("Priority queue is full")
    set elements[size] to (priority, item)
    size += 1
    Call sift_up(size - 1)

  function pop():
    if is_empty():
      raise RuntimeError("Priority queue is empty")
    swap elements[0] with elements[size - 1]
    set  item to elements[size - 1][1]
    size -= 1
    if not is_empty():
      Call sift_down(0)
    return item

  function sift_up(index):
    set parent to (index - 1) // 2
    if index > 0 and elements[index][0] < elements[parent][0]:
      swap elements[index] with elements[parent]
      Call sift_up(parent)

  function sift_down(index):
    set smallest to index
    set left to 2 * index + 1
    set right to 2 * index + 2
    if left < size and elements[left][0] < elements[smallest][0]:
      set smallest to left
    if right < size and elements[right][0] < elements[smallest][0]:
      set smallest to right
    if smallest != index:
      swap elements[index] with elements[smallest]
      sift_down(smallest)
```

Name: Michael McMillan
Student#: 8116775

4. Class: CustomSet

```
class CustomSet:
  function __init__():
    initialize items as empty dictionary
    set count to 0

  function add(item):
    if not contains(item):
      set items[item] to item
      increment count by 1

  function remove(item):
    if contains(item):
      delete items[item]
      decrement count by 1

  function printItems():
    print items

  function contains(item):
    return item in items

  function __contains__(item):
    return contains(item)

  function get_size():
    return count
```

5. Dijkstra's algorithm for shortest path:

```
function dijkstra(graph, start_vertex, goal_vertex):
  set num_vertices to the length of the graph
  initialise distances as an array of infinity with size num_vertices
  initialise parent as an array of -1 with size num_vertices
  set distances[start_vertex] to 0
  initialise pq as new PriorityQueue(num_vertices)
  push (0, start_vertex) into pq

  While not pq.is_empty():
    Set  current_vertex to pq.pop()
    if current_vertex == goal_vertex:
      break
    For each (neighbour, weight) in graph[current_vertex]:
      Set new_distance to distances[current_vertex] + weight
      if new_distance < distances[neighbor]:
        set distances[neighbor] to new_distance
        set parent[neighbor] to current_vertex
        push(new_distance, neighbor) into pq

  initialise path as an empty list
  set current to goal_vertex
```

Name: Michael McMillan
Student#: 8116775

```
  while current != -1:
    append (current + 1) into path  // Convert to 1-based index
    set current to parent[current]
  reverse path
  return path, distances[goal_vertex]
```

6. DFS-based algorithm for longest path:

```
function dfs(graph, current_vertex, goal_vertex, visited, path, max_path, max_length, current_length):
  add current_vertex to visited
  append current_vertex to path
  if current_vertex == goal_vertex:
    if current_length > max_length[0]:
      set max_length[0] to current_length
      clear max_path
      copy path to max_path
  else:
    For each (neighbour, weight) in graph[current_vertex]:
      If neighbor not in visited:
        Call dfs(graph, neighbor, goal_vertex, visited, path, max_path, max_length, current_length +
weight)
  remove current_vertex from path
  remove current_vertex from visited

function find_longest_path(graph, start_vertex, goal_vertex):
  set visited to a new CustomSet()
  set max_path to an empty list
  set max_length to a list containing [0]
  call dfs(graph, start_vertex, goal_vertex, visited, [], max_path, max_length, 0)
  convert max_path to 1-based indexing
  return max_path, max_length[0]
```

Name: Michael McMillan
Student#: 8116775

7. Main function:

```
function main():
  prompt for filename
  try:
    set graph, vertices, start_vertex, goal_vertex to read_file(filename)
  except Exception as e:
    print "Error:", e
    return -1

  print "Number of vertices:", length of vertices
  print "Number of edges:", length of graph
  print "Start vertex:", start_vertex + 1
  print "Goal vertex:", goal_vertex + 1

  set euclidean_dist to euclidean_distance(vertices[start_vertex], vertices[goal_vertex])
  print "Euclidean distance between", start_vertex + 1, "and", goal_vertex + 1, ":", euclidean_dist

  set shortest_path, shortest_length to dijkstra(graph, start_vertex, goal_vertex)
  print "Shortest path:", join(shortest_path with " -> ")\
  print "Shortest path length:", shortest_length

  set longest_path, longest_length to find_longest_path(graph, start_vertex, goal_vertex)
  print "Longest path:", join(longest_path with " -> "
  print "Longest path length:", longest_length

  return 0

if __name__ == "__main__":
  call main()
```

Name: Michael McMillan
Student#: 8116775

Complexity Analysis

1. **Priority Queue:**
   - **Push Operation**
     - The push operation appends the new elements and then sifts up to maintain heap properties.
     - In a binary heap, siftup takes $O(log\ n)$, where n is the number of elements.
     - So, the complexity of push is $O(log\ n)$.
   - **Pop Operation**
     - The pop operation involves swapping the root with the last element, removing the last element, and then performing siftdown.
     - Siftdown also takes $O(\log n)$
     - Hence, the complexity of pop is $O(\log n)$
     - Overall, for k operations (insertions and deletions), the complexity would be $O(k \log n)$.
2. **Read the Graph (function read_file):**
   - The input graph has n vertices and m edges.
   - **Reading vertices:** Parsing the list of vertices requires reading n lines, so the complexity is $O(n)$.
   - **Reading edges:** For m edges, parsing takes $O(m)$.
   - So, reading the graph takes $O(n + m)$.
3. **Dijkstra's Algorithm (Function Dijkstra):**
   - Dijkstra's algorithm is run with a priority queue.
   - **Initialisation:** The distance array is initialised in $O(n)$.
   - **Main loop:** Each vertex is processed at most once, and all its neighbours (edges) are checked for each vertex. A push-and-pop operation is performed on the priority queue for each edge.
     - For each edge, an update in the priority queue costs $O(\log n)$And there are mEdges.
     - Therefore, the total of Dijkstra's algorithm is $O((n + m) \log n)$.
4. **Depth-First Search (DFS) for Longest Path (Function dfs):**
   - DFS explores every path from the start vertex to the goal vertex.
   - **Worst case:** The DFS will traverse every edge and vertex, exploring all possible paths.
   - The DFS complexity is $O(n + m)$, Each vertex and edge is visited once.
5. **Euclidean Distance Calculation (Function euclidean_distance):**
   - This function computes the distance between two vertices, which takes constant time, $O(1)$.
6. **Overall Complexity:**
   - **Dijkstra's Algorithm:** The dominant term in the complexity is $O((n + m) \log n)$, Which is the cost of finding the shortest path.
   - **DFS for Longest Path:** DFS runs in $O(n + m)$, Which is less costly than Dijkstra's algorithm.
   - Therefore, the overall time complexity is $O((n + m) \log n)$, Dominated by Dijkstra's algorithm.

Data Structures List

1. **Priority Queue (PriorityQueue class)**
   - Data Structure: A binary heap implemented using a list
   - **Reason for Use:**
     - Dijkstra's algorithm requires a priority queue to efficiently extract the node with the smallest tentative distance and update the node priorities as distances are recalculated.

Name: Michael McMillan
Student#: 8116775

    - A binary heap provides logarithmic time complexity for both insertions (push) and deletions (pop), making it a good choice for this task.
    - This is crucial for maintaining the performance of Dijkstra's algorithm, which involves frequent queue operations.

2. **List (for storing elements in the priority queue)**
    - **Data Structure:** Python's list (used inside the PriorityQueue class).
    - **Reason for Use:**
        - Lists store the elements (pairs of priority and items) in the priority queue.
        - Python's list allows dynamic resizing and provides easy access to elements for heap operations like sift up and sift down, which are necessary to maintain the heap property.
3. **Dictionary (graph):**
    - **Data Structure:** Python's dict, where the keys are vertex identifiers, and the values are lists of edges (represented as tuples of neighbour vertex and weight).
    - **Reason for Use:**
        - A dictionary allows for efficient lookups when accessing neighbours of a given vertex.
        - The adjacency list representation is well-suited for sparse graphs (where the number of edges is much smaller than the maximum number of edges, $\frac{n(n-1)}{2}$ in an undirected graph). It efficiently stores edges and provides fast access to a vertex's neighbours.
4. **Dictionary (distances)**
    - **Data Structure:** Python's dict, where the keys are vertex identifiers, and the values are the minimum distances from the start vertex.
    - **Reason for Use:**
        - Used to store the shortest distances from the start vertex to each other vertex during Dijkstra's algorithm.
        - The dictionary allows for $O(1)$ lookups and updates when the algorithm computes the new distances for vertices.
5. **Dictionary (parent)**
    - **Data Structure:** Python's dict, where the keys are vertex identifiers, and the values are the parent vertices in the shortest path tree.
    - **Reason for Use:**
        - This structure tracks the parent of each vertex to reconstruct the shortest path once the algorithm terminates.
        - Each vertex's parent can be easily accessed and updated using the dictionary during Dijkstra's algorithm, providing efficient storage and retrieval.
6. **List (for neighbours in graph and path)**
    - **Data Structure:** Python's list, used to store neighbours of each vertex in the adjacency list (graph) and paths in both Dijkstra's and DFS algorithms.
    - **Reasons for Use:**
        - Lists store the neighbours of each vertex because they allow efficient iteration through the vertex when exploring the graph.
        - Lists are also used to store the shortest and longest paths because they are dynamic and allow for easy appending and traversal.
7. **Set (visited)**
    - **Data Structure:** Python's set tracks visited vertices in the DFS for the longest path.
    - **Reason for Use:**

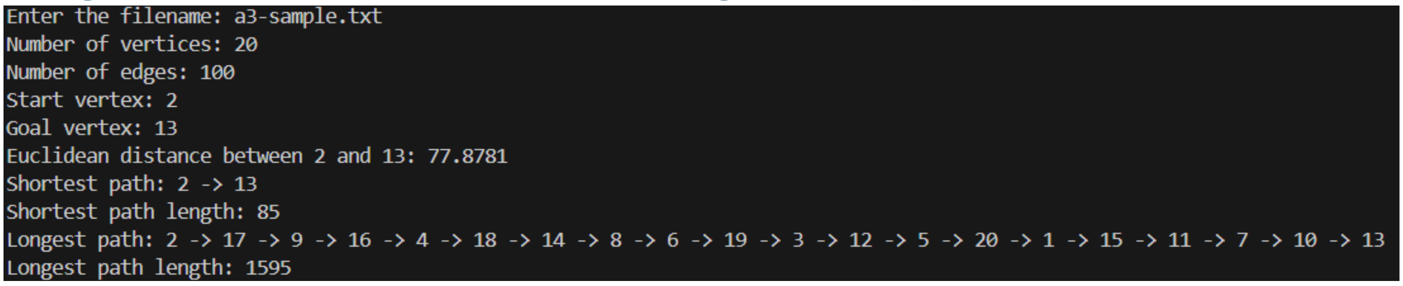
    - A set is used because it provides $O(1)$ time complexity for insertions and membership checks, making it efficient for determining if a vertex has been visited during the DFS.

8. **Tuple (vertices and edges in the graph)**
   - **Data Structure:** Python's tuple stores vertex coordinates (in vertices) and represents edges (as (neighbour, weight) pairs).
   - **Reason for Use:**
     - Tuples are used to store the coordinates of each vertex because the coordinates are immutable once assigned, and tuples are a lightweight and space-efficient data structure for fixed-size collections.
     - Tuples are also used for edges in the adjacency list, as they allow simple grouping of a vertex and its associated weight, which remains unchanged during the graph traversal.
9. **Integers (n, m, vertex IDs, and edge weights)**
   - **Data Structure:** Python's int is used for storing the number of vertices (n), the number of edges (m), vertex IDs, and edge weights.
   - **Reason for Use:**
     - Integers are used to represent values like vertex identifiers and edge weights. These values are essential for indexing the graph, performing arithmetic operations, and comparisons during the execution of Dijkstra's algorithm and DFS.
10. **Float (euclidean_distance)**
    - **Data Structure:** Python's float stores the Euclidean distance between two vertices.
    - **Reason for Use:**
      - Since the Euclidean distance between two points is an actual number, it's appropriate to use floating-point values to store and calculate this value.

Summary of Reasons for Data Structure Choices:

- **Efficiency:** Priority queue for logarithmic time operations, dictionaries for constant-time lookups, sets for fast membership checks, and lists for dynamic storage.
- **Suitability for the Algorithm:** The use of adjacency lists for sparse graphs, priority queues for Dijkstra's shortest path algorithm, and DFS for exploring all paths.
- **Ease of Implementation:** Python's built-in data structures, such as dict, list, and set, provide efficient storage and access while being easy to use with a clear and concise API.

Program Execution Snapshot/Program Outputs



```
Enter the filename: a3-sample.txt
Number of vertices: 20
Number of edges: 100
Start vertex: 2
Goal vertex: 13
Euclidean distance between 2 and 13: 77.8781
Shortest path: 2 -> 13
Shortest path length: 85
Longest path: 2 -> 17 -> 9 -> 16 -> 4 -> 18 -> 14 -> 8 -> 6 -> 19 -> 3 -> 12 -> 5 -> 20 -> 1 -> 15 -> 11 -> 7 -> 10 -> 13
Longest path length: 1595
```

Name: Michael McMillan
Student#: 8116775