Project Name: Marathon

Description:

We have a marathon that involves running from the Start town to the Finish town. The marathon consists of several towns connected by paved roads. Each town has a distance associated with it. The goal is to find the optimal time and the shortest path traveled to complete the marathon from the start town to the finish town.

Rules:

- The Average Speed of Running over paved roads: is 17km/h

Prerequisites:

Visual Studio Code (VS Code) with Java plugin.

Installation:

Setting up Visual Studio Code:

1. Download Visual Studio Code from here:
   (URL: https://code.visualstudio.com/Download)
2. set it up for Java. Download the Coding Pack.
   (URL: https://code.visualstudio.com/docs/java/java-tutorial#:~:text=Getting%20Started%20with%20Java%20in%20VS%20Code%201,debugging%20your%20program%20...%206%20More%20features%20)

Algorithm:

1. Read the input data file "Marathon.csv" containing town details and the distance between the towns.
2. Create a graph data structure to represent the towns and their connections. Each town will be a node, and the connections will be edges with corresponding distances.
3. To find the shortest path from the starting town (S) to the finish town (F) in the graph. [1]
   A. Initialize a queue to store the nodes based on their respective distances.
   B. Initialize an array called distance to store the shortest distance from the S to each town.
   C. Initialize the distance array with infinity values except for the start town (set its distance as 0).
   D. Add the start town to the queue with its distance.
      1. Remove the node with the minimum distance from the queue.
         a) For each neighboring node of the current node:
         b) Add the distance from the S to the neighboring node via the current node.
         c) If this distance is shorter than the previously recorded distance, update the distance array and add the neighboring node to the queue.
      2. Repeat the process until the queue is empty.
4. Keep track of the towns visited and distances covered until reaching F.

- Initialize the total distance and time variables as 0.
  - Iterate over the shortest path, starting from the start town:
    - If the current node is a town
    - Calculate the time taken to run the distance to this town using the average speed on paved roads of 17km/h.
    - Update the total distance and time variables.
  - If the current node is the finish town (F), break the loop.

5. Output the towns taken from the start to the finish line, the total distance taken, and the time taken.

Assumptions:

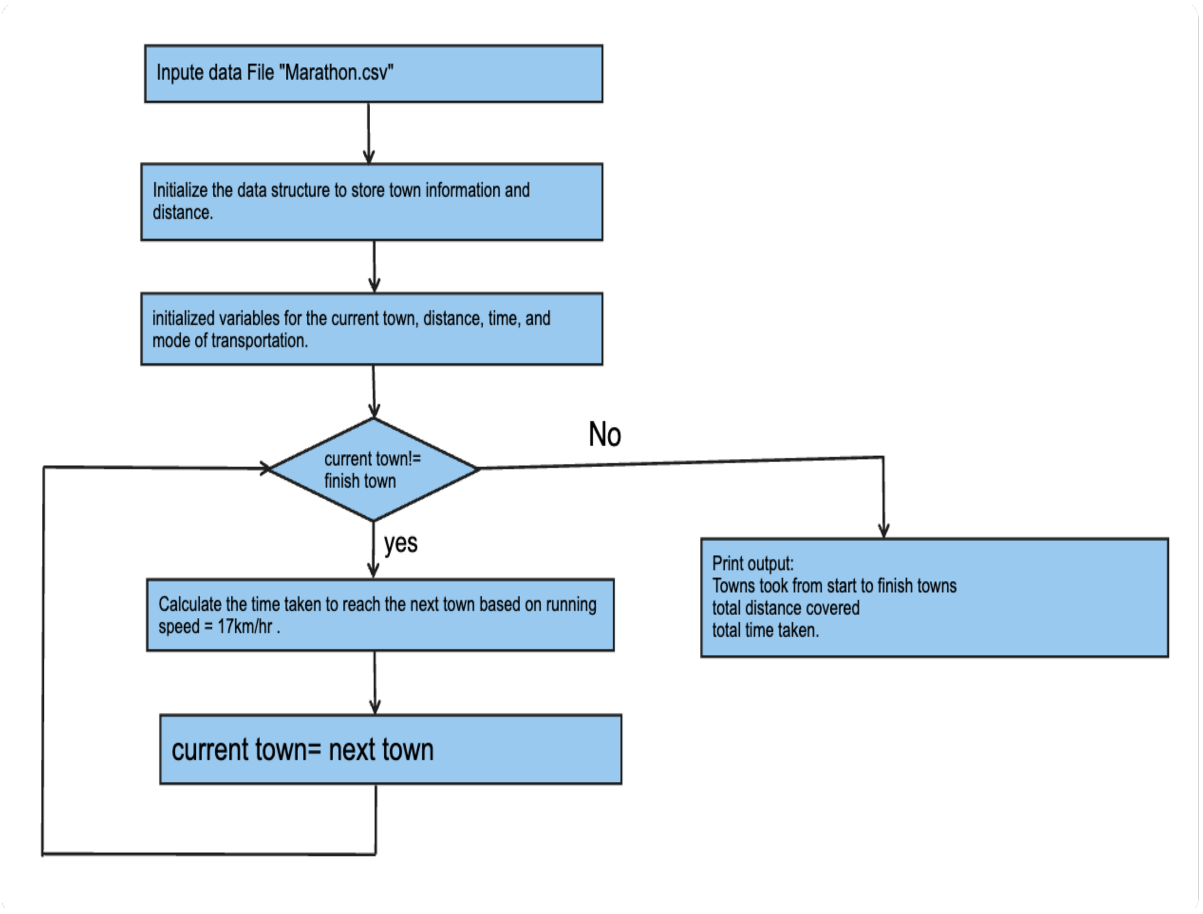
1. Consider the first cell data as the Source town in the CSV file.
2. Consider the last cell data as the Destination town in the CSV file.
3. Assume unconnected paths as 0.

Example:

T1	3	T2
T1	2	T3
T2	5	T4
T2	2	T3
T4	1	T5

- Consider T1 as the Source town and T5 as the destination town.
- Consider the distance between T3 and T4 as 0 km.

Flow Diagram:


Inpute data File "Marathon.csv"
Initialize the data structure to store town information and distance.
initialized variables for the current town, distance, time, and mode of transportation.
current town!= finish town
No
yes
Calculate the time taken to reach the next town based on running speed = 17km/hr .
current town= next town
Print output:
Towns took from start to finish towns
total distance covered
total time taken.

Pseudo Code:

```
class Marathon {
   private Map<String, List<Edge>> routemap;

   class Edge {
      String destination;
      double distance;

      Edge(String destination, double distance) {
         this.destination = destination;
         this.distance = distance;
      }
   }

   class Node {
      String town;
      double distance;

      Node(String town, double distance) {
         this.town = town;
         this.distance = distance;
      }
   }

   Marathon() {
      routemap = new HashMap<>();
   }

   void addEdge(String source, double distance, String destination) {
      // Add edge to the routemap connecting two towns within a distance.
      // If the source town does not exist in the routemap, add it with an empty list of edges.
      // Add the new edge to the list of edges for the source town.
      // Do the same for the destination town.
   }

   List<String> findoptimalPath(String starttown, String finishtown) {
      // Implement Dijkstra's algorithm to find the optimal path and time from the starting town
to the finishing town.
      // Create a PriorityQueue of Nodes based on their distance from the starting town.
      // Initialize distances and previous maps for each town.
      // Add the starting town to the PriorityQueue and set its distance to 0.
      // While the PriorityQueue is not empty:
      // Extract the Node with the minimum distance from the PriorityQueue.
      // If the current town is the finishing town, break the loop.
      // For each neighbor (Edge) of the current town:
      //   Calculate the new distance to the neighbor (considering the average_speed).
      //  If the new distance is smaller than the current distance to the neighbor:
      //    Update the distances and previous maps.
      //    Add the neighbor to the PriorityQueue with its new distance.
      // Reconstruct the optimal path from the starting town to the finishing town using the
previous map.
      // Return the optimal path as a list of towns and print the optimal time to travel from the
starting town to the finishing town.
      // If the optimal time is infinite, return null.
   }

   boolean isTownValid(String town) {
      // Check if a given town is present in the routemap.
      // Return true if the town exists, false otherwise.
   }

   void readFromCSV(String filepath) throws IOException {
      // Read data from a CSV file and populate the routemap using the addEdge() method.
      // Open the CSV file and read each line.
      // Split the line into source town, distance, and destination town.
      // Convert the distance to a double.
      // Call addEdge() with the source, distance, and destination towns to add the edge to the
routemap.
   }

   public static void main(String[] args) {
      // Create a Marathon solver instance.
      // Read data from a CSV file using the readFromCSV() method.
      // If the input towns are not valid, print an error message and return it.
      // Find the optimal path using findoptimalPath() method.
      // If the optimal path is null, print a message that the start and finish towns are
disconnected.
      // Otherwise, print the list of towns traveled for the optimal time.
   }
}
```

Explanation:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Scanner;
```

These are the import statements. They import various classes from the java.io and java.util packages that are used in the program, such as for file reading, data structures, and user input handling.

```
public class Marathon {

// Average speed of Running over paved roads: 17km/h
    private static final double average_speed = 17.0;
    private Map<String, List<Edge>> routemap;
    private static String firstTown;
    private static String lastTown;
```

The Marathon class contains a private static constant average_speed, which represents the average speed of running over paved roads in kilometers per hour. The routemap is a private instance variable of the class, which is a Map that stores a list of Edge objects for each town. Here the keys are town names (strings), and the values are lists of Edge objects. The Edge objects represent connections (edges) between towns in the graph.

```
/* Edge class Represents an edge in the graph, containing the destination town and the distance
to that town. */

private static class Edge {
    String destination;
    double distance;
    Edge(String destination, double distance) {
        this.destination = destination;
        this.distance = distance;
    }
}
```

The inner Edge class represents an edge in the graph, with a destination town and a distance to that town. It has a constructor that takes the destination town and the distance as parameters and initializes the instance variables.

```
public Marathon() {
    routemap = new HashMap<>();
}
```

This is the constructor for the Marathon class. It initializes the routemap instance variable as an empty HashMap.

```
public void addEdge(String source, double distance, String destination) {

// Adds values into ArrayList with source as key when the key is NULL.
    routemap.putIfAbsent(source, new ArrayList<>());
    routemap.get(source).add(new Edge(destination, distance));

    // Assuming unconnected paths as 0 distance.
    routemap.putIfAbsent(destination, new ArrayList<>());
    routemap.get(destination).add(new Edge(source, distance));
}
```

The addEdge method is used to add an edge to the routemap (graph) connecting two towns within a given distance. It takes the source town, distance, and destination town as parameters. The method first checks if the source town already exists in the routemap using “putIfAbsent”. If not, it adds the source town as a key and an empty ArrayList as its value. Then, it adds the Edge representing the destination town and distance to the ArrayList associated with the source town. This process is repeated for the destination town as well to account for the undirected nature of the graph.

```
public List<String> findoptimalPath(String starttown, String finishtown) {
    // ...
}
```

This method, findoptimalPath, calculates the optimal path and time to travel from the start town to the finish town using Dijkstra's algorithm. It returns a list of towns representing the optimal path. The algorithm keeps track of distances from the start town to each town using a “distances” map and the previous town visited on the optimal path using the “previous” map. It uses a priority queue (pq) to efficiently select the next town with the smallest distance for exploration.

```
private static class Node {
    String town;
    double distance;

    Node(String town, double distance) {
        this.town = town;
        this.distance = distance;
    }
}
```

This is another inner class Node, which represents a node in the graph during Dijkstra's algorithm. Each Node contains a town and the current distance from the starting town. The distance field is used to keep track of the current shortest distance from the starting town to this specific town.

```
private boolean isTownValid(String town) {
    return this.routemap.containsKey(town);
}
```

This is a private method, isTownValid, which checks if a given town is present in the routemap (graph). It returns “true” if the town is found, otherwise “false”.

```
public static void main(String[] args) {
    Marathon solver = new Marathon();
    try {
        solver.readFromCSV("data3.csv");
    } catch (IOException e) {
        System.out.println("Failed to read the CSV file.");
        return;
    }
    // ...
}
```

The main method is the entry point of the program. It creates a new instance of the Marathon class called solver. It then calls the readFromCSV method of the solver to populate the routemap with data read from a CSV file (named "Marathon_data.csv" in this case). If the file reading fails, it catches an “IOException” and displays an error message.

```
if (!(solver.isTownValid(starttown) && solver.isTownValid(finishtown))) {
    System.out.println("Input Start or End town is invalid");
    return;
}
```

Here, the code checks if both the start and finish towns are valid (i.e., present in the routemap). If either of them is invalid, it displays an error message and terminates the program.

```
List<String> optimalPath = solver.findoptimalPath(starttown, finishtown);
if (optimalPath == null) {
    System.out.println("Start and finish towns are not connected");
} else {
    System.out.println("Towns traveled to get Optimal time: " + optimalPath);
}
```

This part of the main method calls the findoptimalPath method with the provided starttown and finishtown. It stores the result in the optimalPath variable, which is a list of towns representing the optimal path. If the optimalPath is null, it means that the start and finish towns are not connected, and it displays an appropriate message. Otherwise, it prints the list of towns representing the optimal path.

```
public void readFromCSV(String filepath) throws IOException {
    BufferedReader file = new BufferedReader(new FileReader(filepath));
    // ...
}
```

The readFromCSV method reads data from a CSV file specified by the filepath parameter and populates the routemap (graph) using the addEdge method. It throws an IOException if there is an issue reading the file.

Code:

```
import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Comparator;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.PriorityQueue;

import java.util.Scanner;


public class Marathon

{

// Average speed of Running over paved roads: 17km/h

private static final double average_speed = 17.0;

   private Map<String, List<Edge>> routemap;

   private static String firstTown;

   private static String lastTown;

   /*Edge class Represents an edge in the graph, containing the destination town and the
distance to that town. */

   private static class Edge

   {

      String destination;

      double distance;
```

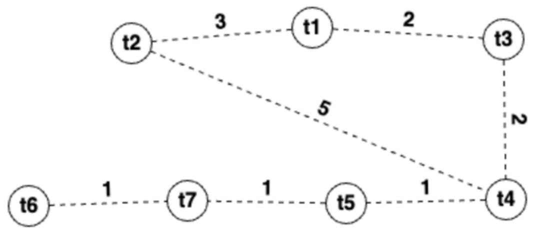
```
        Edge(String destination, double distance)

        {

            this.destination = destination;

            this.distance = distance;

        }

    }


    public Marathon()

    {

        routemap = new HashMap<>();

    }


    /* addEdge method adds an edge to the routemap (graph) connecting two towns with a
given distance. */

    public void addEdge(String source, double distance, String destination)

    {

            // Adds values into ArrayList with source as key when key is NULL.
            routemap.putIfAbsent(source, new ArrayList<>());

        routemap.get(source).add(new Edge(destination, distance));

        // Assuming unconnected paths as 0 distance.

        routemap.putIfAbsent(destination, new ArrayList<>());

        routemap.get(destination).add(new Edge(source, distance));

    }

    /* findoptimalPath method calculates the optimal path and time to travel from the starting
town to the finishing town using Dijkstra's algorithm. */
```

```
  public List<String> findoptimalPath(String starttown, String finishtown)
  {
    Map<String, Double> distances = new HashMap<>();
    Map<String, String> previous = new HashMap<>();
    PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingDouble(n ->
n.distance));

    for (String town : routemap.keySet()) {
      if (town.equals(starttown)) {
        distances.put(town, 0.0);
        pq.offer(new Node(town, 0.0));
      } else {
        distances.put(town, Double.POSITIVE_INFINITY);
        pq.offer(new Node(town, Double.POSITIVE_INFINITY));
      }
      previous.put(town, null);
    }

    while (!pq.isEmpty()) {
      Node currentnode = pq.poll();
      String currenttown = currentnode.town;

      if (currenttown.equals(finishtown)) {
        break; // found the finish town, exit the loop.
      }
```

```
        if (distances.get(currenttown) < currentnode.distance) {
            continue; // skip this iteration if already found a shorter distance to this town
        }
        for (Edge edge : routemap.get(currenttown)) {
            double newdistance = distances.get(currenttown) + edge.distance / average_speed;
            if (newdistance < distances.get(edge.destination)) {
                distances.put(edge.destination, newdistance);
                previous.put(edge.destination, currenttown);
                pq.offer(new Node(edge.destination, newdistance));
            }
        }
    }
    List<String> optimalpath = new ArrayList<>();
    //double optimalDistance = 0.0;
    String currTown = finishtown;

    while (currTown != null) {
        optimalpath.add(0, currTown);
        currTown = previous.get(currTown);
    }
    System.out.println("Optimal Time: "+String.format("%.4f",distances.get(finishtown))+" hours");
   System.out.println("Total Distance Traveled: 
"+String.format("%.4f",distances.get(finishtown)*average_speed)+" kms");
    if(distances.get(finishtown).equals(Double.POSITIVE_INFINITY)) {
        return  null;
    }

    return optimalpath;

}

/* Node class represents a node in the graph during Dijkstra's algorithm. It contains the
town and the current distance from the starting town. */

private static class Node

{

    String town;

    double distance;

    Node(String town, double distance)

    {

        this.town = town;

        this.distance = distance;

    }

}

// Methode isTownValid Checks if a given town is present in the routemap (graph).

private boolean isTownValid(String town) {

    return this.routemap.containsKey(town);

}

public static void main(String[] args)

{

    Marathon solver = new Marathon();

    try {

        solver.readFromCSV("Marathon_data2.csv"); // Enter the file name

    } catch (IOException e) {
        System.out.println("Failed to read the CSV file.");
        return;
    }
    //global set in readCSV
    String starttown = firstTown;
    System.out.println("Start town is " + starttown);
    String finishtown = lastTown;
    System.out.println("Finish town is " + finishtown);

    if(!(solver.isTownValid(starttown)&&solver.isTownValid(finishtown))) {
        System.out.println("Input Start or End town is invalid");
        return;
    }
    List<String> optimalPath = solver.findoptimalPath(starttown, finishtown);
    if(optimalPath == null) {
        System.out.println("Start and finish towns are not connected");
    } else {
        System.out.println("Towns traveled to get Optimal time: " + optimalPath);
    }
}
/* readFromCSV method Reads data from a CSV file and populates the routemap (graph)
using the addEdge() method */
public void readFromCSV(String filepath) throws IOException
{
    BufferedReader file = new BufferedReader(new FileReader(filepath));
    String line = "";
    boolean firstLine = true;
    while ((line = file.readLine()) != null) {
        String[] towns = line.trim().split(",");
        String source = towns[0].trim();
        double distance = Double.parseDouble(towns[1]);
        String destination = towns[2].trim();
        addEdge(source, distance, destination);
        if(firstLine) {
            firstTown = source;
            firstLine = false;
        }
        lastTown = destination;
    }
}
}
```

Marathon_Data2.csv file:

t1	3	t2
t1	2	t3
t2	5	t4
t3	2	t4
t4	1	t5
t6	4	t7
t5	1	t7



Output:

```
Start town is t1
Finish town is t7
Optimal Time: 0.3529 hours
Total Distance Traveled: 6.0000 kms
Towns traveled to get Optimal time: [t1, t3, t4, t5, t7]
```

References:

[1]:https://www.analyticssteps.com/blogs/dijkstras-algorithm-shortest-path-algorithm
[2]: https://www.javatpoint.com/dijkstra-algorithm-java

Contact:

Email id: shanmukha.tippavajhala@student.ufv.ca