Data Structures and Algorithms Assignment 2

Report

Semester 1, 2023

Name – Manpriya Laksahan Pathirana

Curtin ID – 20922993

Index

1. User guide

Unit test harness
task :1 =GraphConstruction.java
task :2 =GraphTraversal.java
task :3 =GraphOperations.java
task :4 =HashTableImplementation.java
task :5 =HeapUsage.java
task :6 =FlightPathOptimizer.java
task :7 =BushfireRiskAnalyzer.java

2. Description of classes

2.0)Unit Test Harness
2.1)GraphConstruction.java
2.2)GraphTraversal.java
2.3)GraphOperations.java
2.4)HashTableImplementation.java
2.5)HeapUsage.java
2.6)FlightPathOptimizer.java
2.7)BushfireRiskAnalyzer.java

3. *Testing methodology*

2.1)GraphOperations.java
2.2)HashTableImplementation.java
2.3)BushfireRiskAnalyzer.java

4. *uml Diagram*

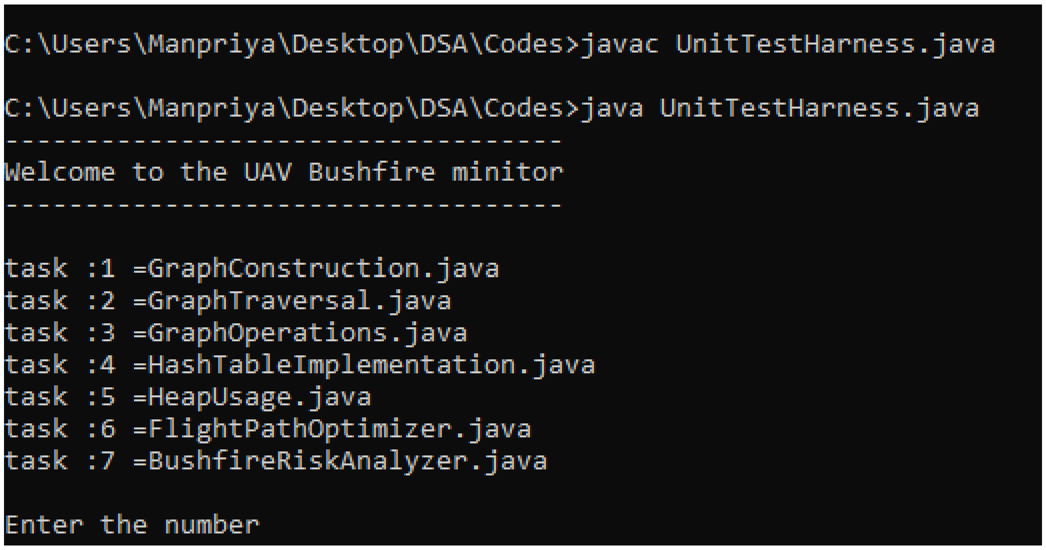
5. Conclusion and Future Work

User Guide

Unit Test Harness

1. Introduction
   This program can run all following java files –
      GraphConstruction.java
      GraphTraversal.java
      GraphOperations.java
      HashTableImplementation.java
      HeapUsage.java
      FlightPathOptimizer.java
      BushfireRiskAnalyzer.java

2. Input Files: "location.txt" , "UAVdata.txt"
3.
4. Program Execution & Output:
5.



```
C:\Users\Manpriya\Desktop\DSA\Codes>javac UnitTestHarness.java

C:\Users\Manpriya\Desktop\DSA\Codes>java UnitTestHarness.java
-----------------------------------
Welcome to the UAV Bushfire minitor
-----------------------------------

task :1 =GraphConstruction.java
task :2 =GraphTraversal.java
task :3 =GraphOperations.java
task :4 =HashTableImplementation.java
task :5 =HeapUsage.java
task :6 =FlightPathOptimizer.java
task :7 =BushfireRiskAnalyzer.java

Enter the number
```

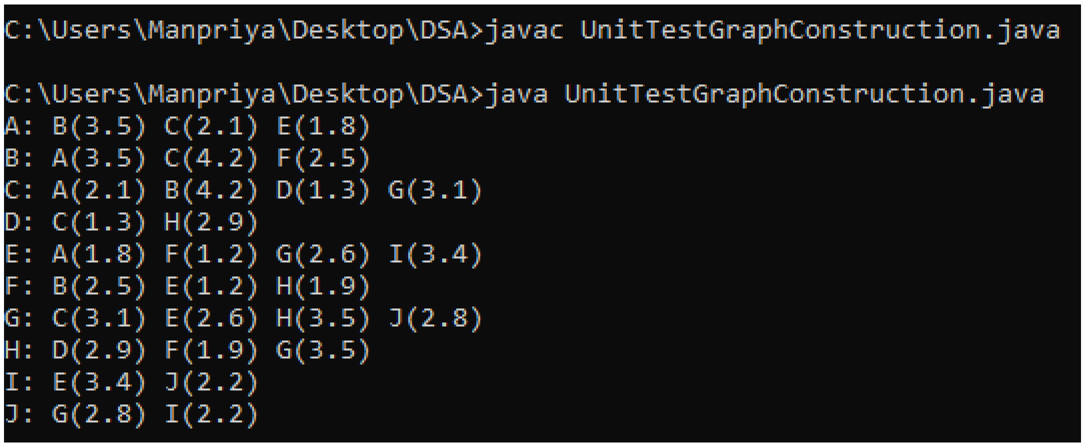
Note: Make sure the input files follow the specified formats ("location.txt" and "UAVdata.txt") and are correctly named in the same directory as the Java file before running the program .

Task :1 =GraphConstruction.java

1. Introduction:
   This is a simple Java program that reads data from a file, constructs a graph, and prints the adjacency list representation of the graph.

2. Input File :   "location.txt"

3. Program Execution & Output:



```
C:\Users\Manpriya\Desktop\DSA>javac UnitTestGraphConstruction.java

C:\Users\Manpriya\Desktop\DSA>java UnitTestGraphConstruction.java
A: B(3.5) C(2.1) E(1.8)
B: A(3.5) C(4.2) F(2.5)
C: A(2.1) B(4.2) D(1.3) G(3.1)
D: C(1.3) H(2.9)
E: A(1.8) F(1.2) G(2.6) I(3.4)
F: B(2.5) E(1.2) H(1.9)
G: C(3.1) E(2.6) H(3.5) J(2.8)
H: D(2.9) F(1.9) G(3.5)
I: E(3.4) J(2.2)
J: G(2.8) I(2.2)
```

Note: Make sure the input file follows the specified format and is correctly named as "location.txt" in the same directory as the Java file before running the program.

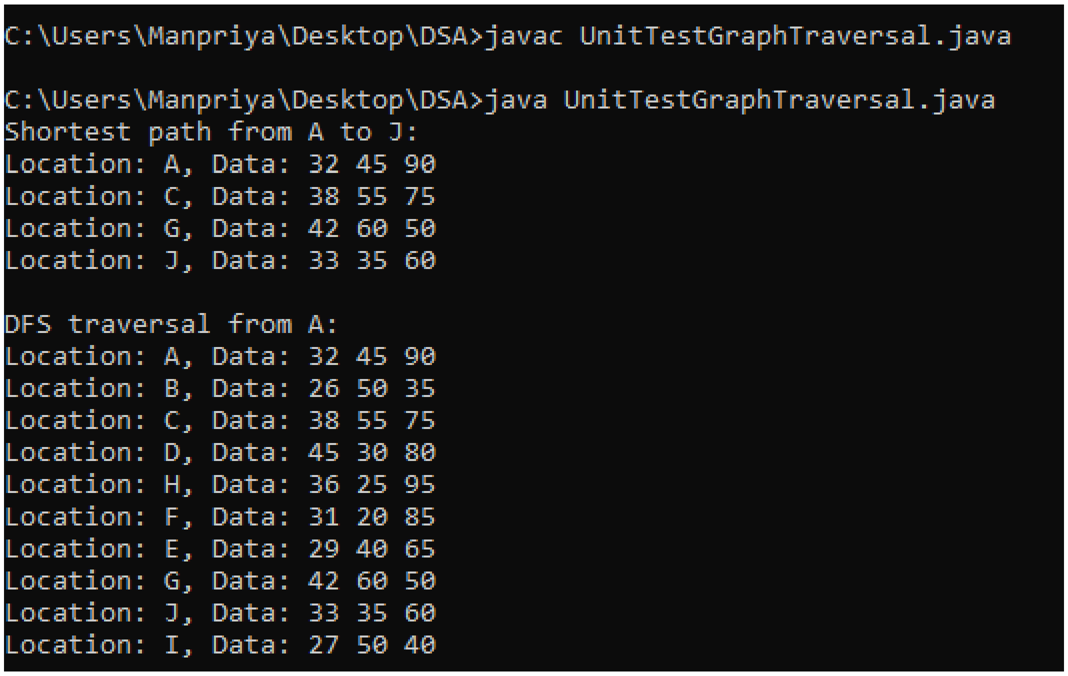
task :2 =GraphTraversal.java

User Guide for GraphTraversal Code:

1. Introduction:
   This Java program that performs graph traversal algorithms, namely Breadth-First Search (BFS) and Depth-First Search (DFS), on a graph represented by adjacency lists. The program reads data from two input files ("location.txt" and "UAVdata.txt"), constructs the graph, and performs BFS and DFS traversal.

2. Input Files: "location.txt" , "UAVdata.txt"

3. Program Execution & Output:



```
C:\Users\Manpriya\Desktop\DSA>javac UnitTestGraphTraversal.java

C:\Users\Manpriya\Desktop\DSA>java UnitTestGraphTraversal.java
Shortest path from A to J:
Location: A, Data: 32 45 90
Location: C, Data: 38 55 75
Location: G, Data: 42 60 50
Location: J, Data: 33 35 60

DFS traversal from A:
Location: A, Data: 32 45 90
Location: B, Data: 26 50 35
Location: C, Data: 38 55 75
Location: D, Data: 45 30 80
Location: H, Data: 36 25 95
Location: F, Data: 31 20 85
Location: E, Data: 29 40 65
Location: G, Data: 42 60 50
Location: J, Data: 33 35 60
Location: I, Data: 27 50 40
```

Note: Make sure the input files follow the specified formats ("location.txt" for graph edges and weights , and "UAVdata.txt" for location-data associations) and are correctly named in the same directory as the Java file before running the program .

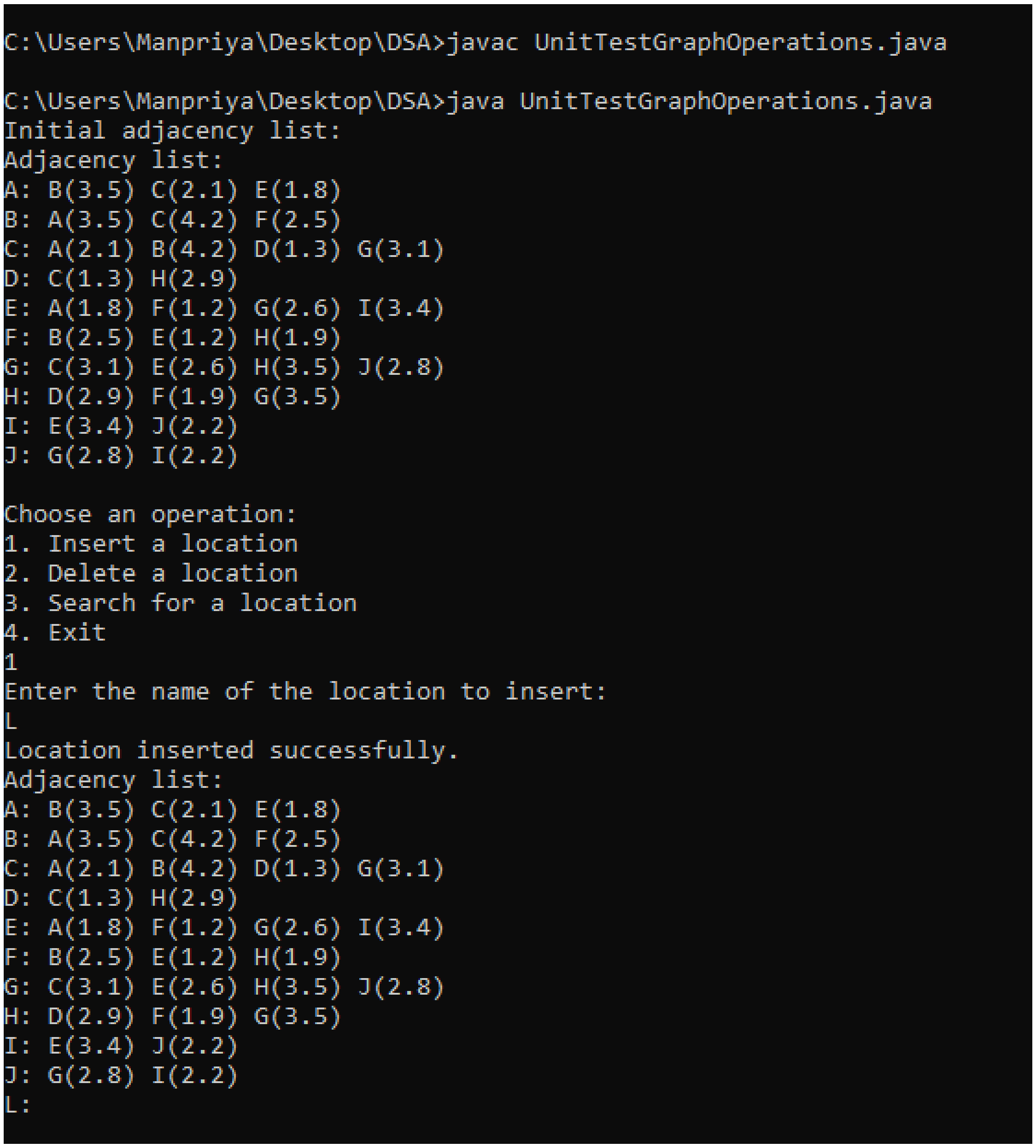
task :3 =GraphOperations.java

User Guide for GraphOperations Code:

1. Introduction:
   This program that allows you to perform basic operations on a graph, such as inserting a location, deleting a location, and searching for a location . The program reads the initial graph structure from an input file ("location.txt") and provides a menu-driven interface for interacting with the graph

2. Input File: "location.txt"

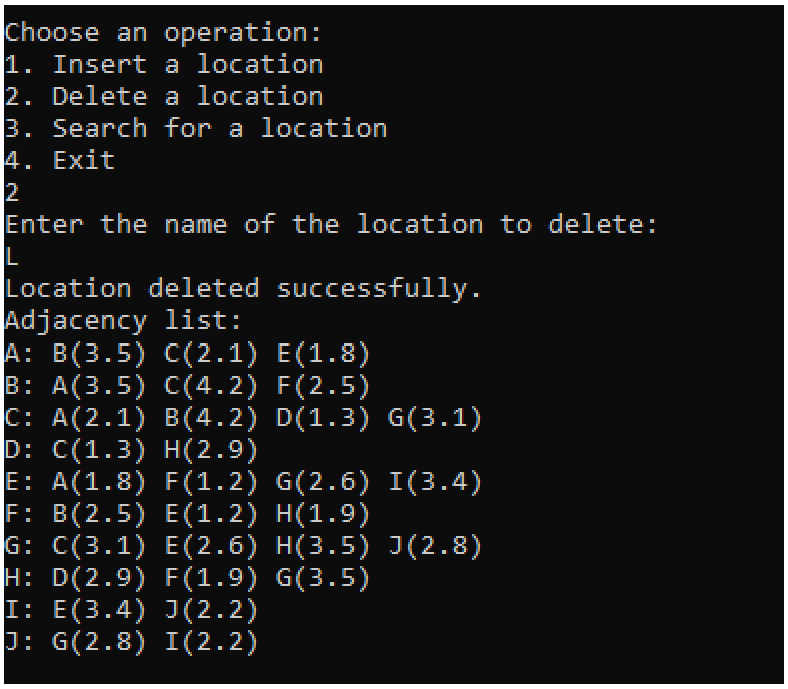
3. Program Execution & out put :



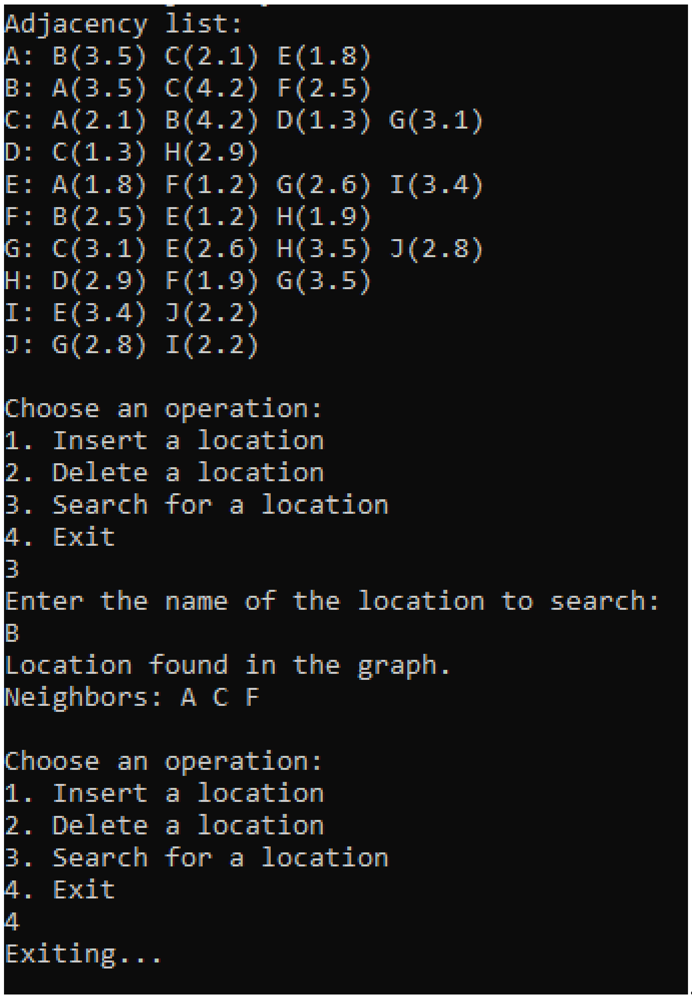
```
C:\Users\Manpriya\Desktop\DSA>javac UnitTestGraphOperations.java

C:\Users\Manpriya\Desktop\DSA>java UnitTestGraphOperations.java
Initial adjacency list:
Adjacency list:
A: B(3.5) C(2.1) E(1.8)
B: A(3.5) C(4.2) F(2.5)
C: A(2.1) B(4.2) D(1.3) G(3.1)
D: C(1.3) H(2.9)
E: A(1.8) F(1.2) G(2.6) I(3.4)
F: B(2.5) E(1.2) H(1.9)
G: C(3.1) E(2.6) H(3.5) J(2.8)
H: D(2.9) F(1.9) G(3.5)
I: E(3.4) J(2.2)
J: G(2.8) I(2.2)

Choose an operation:
1. Insert a location
2. Delete a location
3. Search for a location
4. Exit
1
Enter the name of the location to insert:
L
Location inserted successfully.
Adjacency list:
A: B(3.5) C(2.1) E(1.8)
B: A(3.5) C(4.2) F(2.5)
C: A(2.1) B(4.2) D(1.3) G(3.1)
D: C(1.3) H(2.9)
E: A(1.8) F(1.2) G(2.6) I(3.4)
F: B(2.5) E(1.2) H(1.9)
G: C(3.1) E(2.6) H(3.5) J(2.8)
H: D(2.9) F(1.9) G(3.5)
I: E(3.4) J(2.2)
J: G(2.8) I(2.2)
L:
```



```
Choose an operation:
1. Insert a location
2. Delete a location
3. Search for a location
4. Exit
2
Enter the name of the location to delete:
L
Location deleted successfully.
Adjacency list:
A: B(3.5) C(2.1) E(1.8)
B: A(3.5) C(4.2) F(2.5)
C: A(2.1) B(4.2) D(1.3) G(3.1)
D: C(1.3) H(2.9)
E: A(1.8) F(1.2) G(2.6) I(3.4)
F: B(2.5) E(1.2) H(1.9)
G: C(3.1) E(2.6) H(3.5) J(2.8)
H: D(2.9) F(1.9) G(3.5)
I: E(3.4) J(2.2)
J: G(2.8) I(2.2)
```



```
Adjacency list:
A: B(3.5) C(2.1) E(1.8)
B: A(3.5) C(4.2) F(2.5)
C: A(2.1) B(4.2) D(1.3) G(3.1)
D: C(1.3) H(2.9)
E: A(1.8) F(1.2) G(2.6) I(3.4)
F: B(2.5) E(1.2) H(1.9)
G: C(3.1) E(2.6) H(3.5) J(2.8)
H: D(2.9) F(1.9) G(3.5)
I: E(3.4) J(2.2)
J: G(2.8) I(2.2)

Choose an operation:
1. Insert a location
2. Delete a location
3. Search for a location
4. Exit
3
Enter the name of the location to search:
B
Location found in the graph.
Neighbors: A C F

Choose an operation:
1. Insert a location
2. Delete a location
3. Search for a location
4. Exit
4
Exiting...
```
.

Note: The program assumes the input file "location.txt" follows a specific format and is correctly named in the same directory as the Java file.

task :4 =HashTableImplementation.java

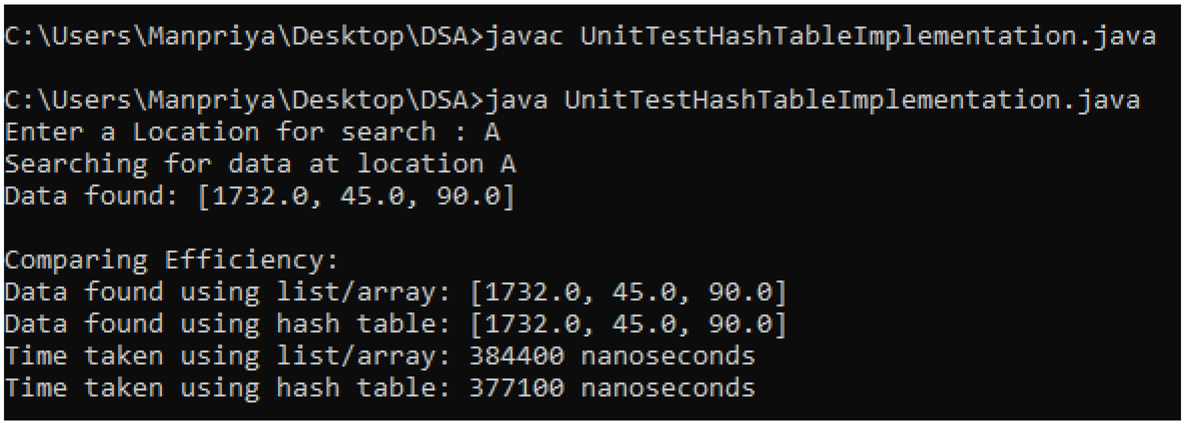
1. Introduction:

This Java program that demonstrates the implementation of a hash table using the Hashtable class. The program reads data from an input file ("UAVdata.txt") and stores it in the hash table. It also provides a method to search for data based on a given location and compares the efficiency of searching through a list/array versus a hash table.

2. Input File: "UAVdata.txt"

This file contains data in the format: <location> <data1> <data2> ... <dataN>. Each line represents a different location with its corresponding data values.

3. Program Execution & output :



```
C:\Users\Manpriya\Desktop\DSA>javac UnitTestHashTableImplementation.java

C:\Users\Manpriya\Desktop\DSA>java UnitTestHashTableImplementation.java
Enter a Location for search : A
Searching for data at location A
Data found: [1732.0, 45.0, 90.0]

Comparing Efficiency:
Data found using list/array: [1732.0, 45.0, 90.0]
Data found using hash table: [1732.0, 45.0, 90.0]
Time taken using list/array: 384400 nanoseconds
Time taken using hash table: 377100 nanoseconds
```

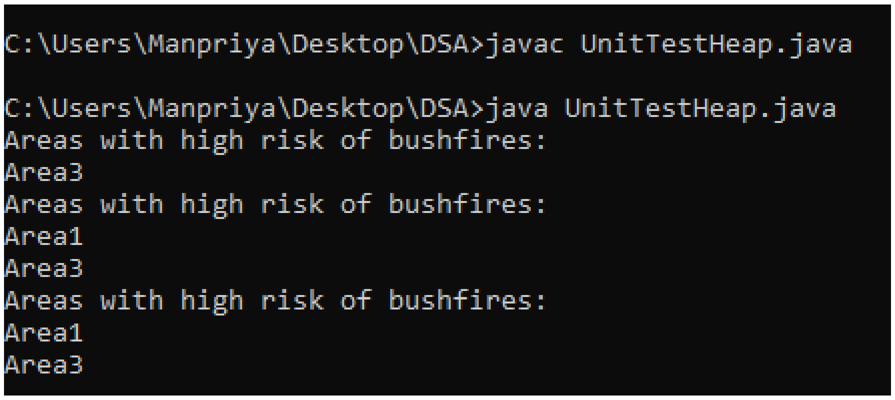
Note: The program assumes the input file "UAVdata.txt" follows a specific format and is correctly named in the same directory as the Java file.

task :5 =Heap.java

1. Introduction:
   This Java program that demonstrates the usage of a heap data structure. It includes a Heap class that implements a max heap, allowing for the insertion, updating, deletion, and printing of areas with high-risk values. The program initializes a heap, inserts areas with their corresponding risk values , performs updates on risk values, deletes an area , and prints the areas with high risk values.

2. Input: The program does not require any external input files. The areas and their corresponding risk values are specified within the code itself.

3. Program Execution & output :



```
C:\Users\Manpriya\Desktop\DSA>javac UnitTestHeap.java

C:\Users\Manpriya\Desktop\DSA>java UnitTestHeap.java
Areas with high risk of bushfires:
Area3
Areas with high risk of bushfires:
Area1
Area3
Areas with high risk of bushfires:
Area1
Area3
```

Note: The program assumes that the provided heap implementation is correct and functioning as expected. The risk threshold for determining high-risk areas is set to a value greater than 40.

task :6 =FlightPathOptimizer.java

1. Introduction:

This Java program that optimizes a flight path based on a graph representation. It uses Dijkstra's algorithm to find the shortest path from a given source location to a destination while avoiding high-risk locations. The program builds a graph from a specified file, performs the flight path optimization, and outputs the optimized flight path.

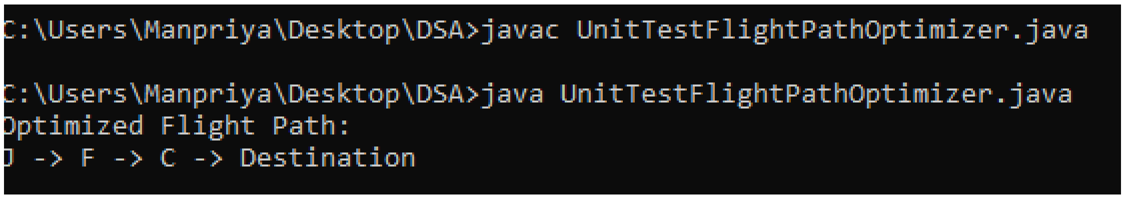
2. Input Files:

The program requires a file containing information about the graph. The file should be in the following format : <start_location> <distance> <time>

Each line represents an edge in the graph, where:

- <start_location> is the starting location of the edge.
- <distance> is the distance of the edge.
- <time> is the time taken to traverse the edge.

3. Program Execution Output :



```
:\Users\Manpriya\Desktop\DSA>javac UnitTestFlightPathOptimizer.java

:\Users\Manpriya\Desktop\DSA>java UnitTestFlightPathOptimizer.java
ptimized Flight Path:
 -> F -> C -> Destination
```

Note: The program assumes that the provided graph file is correctly formatted and contains valid data. Additionally, it assumes that the graph is connected, meaning there is a valid path from the source to any other location in the graph.

task :7 =BushfireRiskAnalyzer.java

1. Introduction:
   The provided code is a Java program that analyzes bushfire risk data collected by Unmanned Aerial Vehicles (UAVs). It allows users to load UAV data files, analyze the data, and obtain optimized flight itineraries for mitigating bushfire risks in high-risk locations.

2. Input Files:

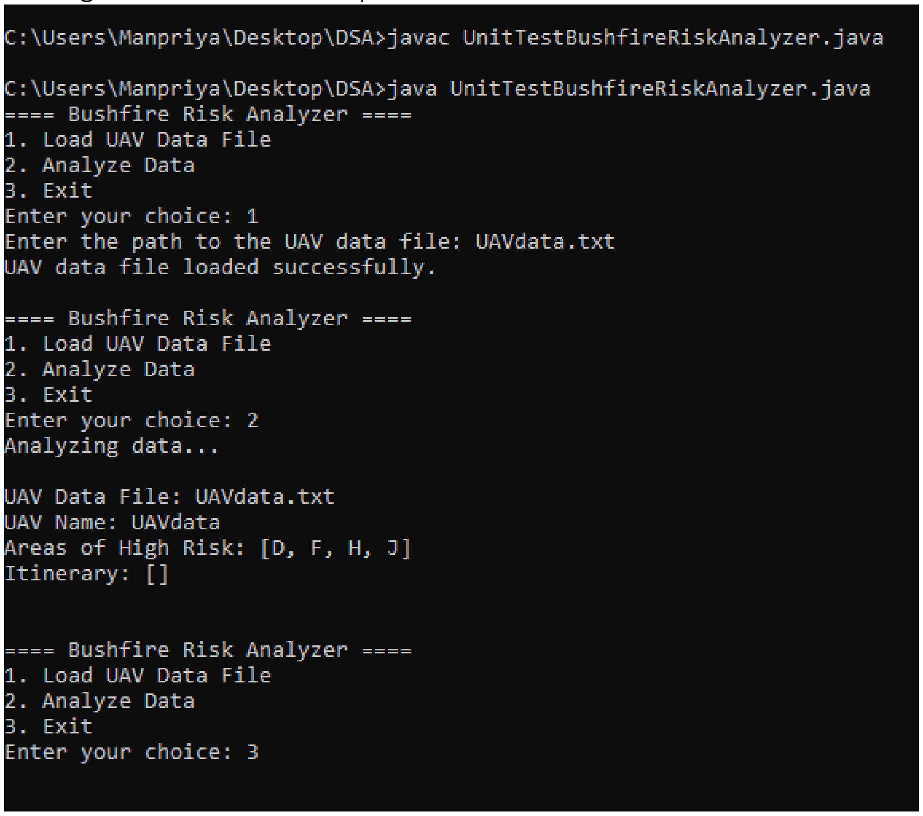
   The program expects UAV data files containing location-specific data in the following format:

 <location> <temperature> <humidity> <wind_speed>

   Each line represents data for a specific location, where:
   - <location> is the name or identifier of the location.
   - <temperature> is the temperature at the location.
   - <humidity> is the humidity level at the location.
   - <wind_speed> is the wind speed at the location.

   When loading a UAV data file, you will be prompted to enter the file path.

3. Program Execution & output :



```
C:\Users\Manpriya\Desktop\DSA>javac UnitTestBushfireRiskAnalyzer.java

C:\Users\Manpriya\Desktop\DSA>java UnitTestBushfireRiskAnalyzer.java
==== Bushfire Risk Analyzer ====
1. Load UAV Data File
2. Analyze Data
3. Exit
Enter your choice: 1
Enter the path to the UAV data file: UAVdata.txt
UAV data file loaded successfully.

==== Bushfire Risk Analyzer ====
1. Load UAV Data File
2. Analyze Data
3. Exit
Enter your choice: 2
Analyzing data...

UAV Data File: UAVdata.txt
UAV Name: UAVdata
Areas of High Risk: [D, F, H, J]
Itinerary: []


==== Bushfire Risk Analyzer ====
1. Load UAV Data File
2. Analyze Data
3. Exit
Enter your choice: 3
```

The program continues to run until the user chooses the "Exit" option.

Note: Ensure that the UAV data files are correctly formatted and contain valid data in order to obtain accurate results.

Description of Classes

Unit Test Harness

The   UnitTestHarness class serves as the entry point for the program. It contains a main method that allows the user to select and execute various unit tests related to UAV (Unmanned Aerial Vehicle) bushfire monitoring .

The class begins by displaying a welcome message and a list of available tasks. The user is prompted to enter a number corresponding to the desired task. After receiving the input, the program uses a `switch` statement to determine which unit test should be executed.

The available tasks include:

1. GraphConstruction.java: Performs unit testing for graph construction functionality.

2. GraphTraversal.java: Executes unit tests related to graph traversal operations.

3. GraphOperations.java: Runs unit tests for graph manipulation and operations.

4. HashTableImplementation.java: Conducts unit testing for the implementation of a hash table.

5. HeapUsage.java: Executes unit tests related to heap data structure usage.

6. FlightPathOptimizer.java: Performs unit testing for flight path optimization algorithms.

7. BushfireRiskAnalyzer.java: Executes unit tests for bushfire risk analysis.

GraphConstruction

1. GraphConstruction (Main Class): This is the main class that contains the main method, serving as the entry point for the program.

2. Map<String, Map<String, Double>> graph: This is a data structure used to represent the graph. It is a nested map where the outer map's keys are the vertices (String) and the corresponding values are inner maps. The inner maps store the neighboring vertices as keys and their corresponding edge weights (Double) as values.

3. BufferedReader: This class is used to read the contents of a file line by line.

4. FileReader: This class is used to read characters from a file.

5. HashMap: This class is used to implement the graph variable. It provides a fast way to store and retrieve data using key-value pairs.

GraphTraversal

1. GraphTraversal (Main Class): This is the main class that contains the main method, serving as the entry point for the program.

2. Map<String, Map<String, Double>> graph: This is a data structure used to represent the graph. It is a nested map where the outer map's keys are the vertices (String) and the corresponding values are inner maps. The inner maps store the neighboring vertices as keys and their corresponding edge weights (Double) as values.

3. Map<String, List<Pair<String, Double>>> adjacencyList: This data structure represents an adjacency list, which provides a list of connected vertices for each vertex in the graph. It is a map where the keys are vertices (String) and the values are lists of pairs. Each pair consists of a connected vertex (String) and the weight of the edge (Double).

4. Map<String, String> dataMap: This map stores additional data associated with each vertex in the graph. The keys are vertices (String), and the values are strings representing the associated data.

5. BufferedReader: This class is used to read the contents of a file line by line.

6. FileReader: This class is used to read characters from a file.

7. Pair<T, U> (Inner Class): This class defines a simple generic pair data structure used to store two values together.

GraphOperations

1. GraphOperations (Main Class): This is the main class that contains the main method, serving as the entry point for the program. It also encapsulates the graph data structure and provides methods for performing various operations on the graph.

2. private static Map<String, Map<String, Double>> graph: This is a class-level variable that represents the graph. It is a nested map where the outer map's keys are the vertices (String), and the corresponding values are inner maps. The inner maps store the neighboring vertices as keys and their corresponding edge weights (Double) as values.

3. BufferedReader: This class is used to read the contents of a file line by line.

4. FileReader: This class is used to read characters from a file.

5. runMenu()method: This method displays a menu of operations and takes user input to execute the chosen operation. It uses a Scanner object to read user input from the command line.

6. insertLocation(Scanner scanner) method: This method prompts the user to enter the name of a location and inserts it into the graph. If the location already exists, it displays a message indicating that it already exists. Otherwise, it adds the location as a vertex in the graph and displays the updated adjacency list.

7. deleteLocation(Scanner scanner) method: This method prompts the user to enter the name of a location and deletes it from the graph. If the location does not exist, it displays a message indicating that it doesn't exist. Otherwise, it removes the location and its associated edges from the graph and displays the updated adjacency list.

8. searchLocation(Scanner scanner) method: This method prompts the user to enter the name of a location and searches for it in the graph. If the location is found, it displays a message indicating its presence and lists its neighboring vertices. If the location is not found, it displays a message indicating its absence.

9. displayAdjacencyList()method: This method displays the current adjacency list representation of the graph. It iterates over each vertex in the graph, retrieves its neighboring vertices and edge weights, and prints them.

HashTableImplementation

1. HashTableimplementation (Main Class): This is the main class that contains the main method, serving as the entry point for the program. It encapsulates the hash table data structure and provides methods for performing various operations on the hash table.

2. private static Map<String, List<Double>> hashTable: This is a class-level variable that represents the hash table. It is implemented using the Hashtable class, where the keys are of type String and the values are lists of Double values.

3. BufferedReader: This class is used to read the contents of a file line by line.

4. FileReader: This class is used to read characters from a file.

5. searchInHashTable(String location) method: This method takes a location as input and returns the list of Double values associated with that location from the hash table. It uses the get method of the Hashtable class to retrieve the data corresponding to the given key.

6. compareEfficiency()method: This method compares the efficiency of searching through a list or array versus using a hash table. It retrieves the list of locations from the hash table, iterates over them, and compares each location with the target search location. If a match is found, it retrieves the associated data. It also retrieves the data directly from the hash table

Heap

1. Heap (Main Class):

This class contains the main method and serves as the entry point for the program. In the main method, an instance of the Heap class called heap is created. The heap object is used to demonstrate various operations on a heap data structure. Operations performed include inserting elements into the heap, updating the risk associated with specific areas, deleting elements from the heap, and printing areas with high risk.

- The results of the operations are displayed by calling methods on the heap object.

2. Heap :

- This class represents a heap data structure . It contains private member variables for maintaining the heap, including the size of the heap, arrays for storing the areas and risks, an array for representing the heap structure, and a map for storing the index of each area in the heap

FlightPathOptimizer

1. FlightPathOptimizer Class:

- This class represents a flight path optimizer that uses a graph data structure to optimize flight paths. It contains a private member variable graph, which is a nested map that stores the graph data. The outer map represents the vertices, and the inner map represents the edges with their associated weights (distance and time).

- The class provides the following methods:

  -FlightPathOptimizer(): Constructor that initializes the graph map as a new empty HashMap.

  - buildGraph(String filePath): Builds the graph by reading data from a file. The file should contain lines with vertex, distance, and time information separated by spaces. The method parses each line and populates the graph map accordingly.

  - optimizeFlightPath(String source, Set<String> highRiskLocations): Optimizes the flight path from a given source location, considering a set of high-risk locations. The method uses Dijkstra's algorithm to find the shortest path from the source to each high-risk location. It returns the optimized flight path as a list of locations.

  - Node (nested class): Represents a node in the graph. It holds the vertex name and the distance from the source.

  - NodeComparator (nested class): Implements a comparator for comparing nodes based on their distances. It is used by the priority queue to prioritize nodes with smaller distances.

  - main(String[] args): The entry point of the program. It creates an instance of FlightPathOptimizer, builds the graph from a file, specifies a source location and a set of high-risk locations, and then optimizes the flight path. The resulting optimized flight path is printed to the console.

BushfireRiskAnalyzer

1. BushfireRiskAnalyzer Class:

- This class represents a bushfire risk analyzer that allows loading UAV (Unmanned Aerial Vehicle) data files and analyzing the data to identify areas of high risk.

- It contains three private member variables: graph, highRiskLocations, and uavDataFiles.

- graph: A nested map that represents the graph data. The outer map represents the locations, and the inner map represents the data attributes (temperature, humidity, and wind speed) associated with each location.

  - highRiskLocations: A set that stores the locations identified as high-risk areas.

  - uavDataFiles: A list that holds the paths of the loaded UAV data files.

- The class provides the following methods:

  - BushfireRiskAnalyzer(): Constructor that initializes the graph, highRiskLocations, and uavDataFiles.

  - run(): Starts the bushfire risk analyzer. It displays a menu with options to load UAV data files, analyze data, or exit the program. It takes user input to perform the chosen action.

  - loadUAVDataFile(Scanner scanner): Loads a UAV data file by prompting the user to enter the file path. It checks if the file exists, adds the file path to uavDataFiles, and displays a success message.

  - analyzeData(): Analyzes the loaded UAV data files by creating instances of FlightPathOptimizer for each file, building the graph, and optimizing flight paths. It displays the analysis results, including the UAV data file path, UAV name, high-risk areas, and optimized itinerary.

  - getUAVNameFromFilePath(String filePath): Extracts the UAV name from the file path by splitting the path and retrieving the filename without the extension.

  - FlightPathOptimizer (nested class): Represents a flight path optimizer specific to the bushfire risk analyzer. It is responsible for building the graph based on the UAV data and optimizing flight paths.

    - buildGraph(String filePath): Reads the UAV data file specified by the file path, extracts location and data attributes (temperature, humidity, and wind speed), checks for high-risk conditions, and populates the graph and highRiskLocations.

    - optimizeFlightPath(String source, Set<String> highRiskLocations): Placeholder method that returns an empty list. It represents the optimization of flight paths, which is not implemented in the provided code.

2. FlightPathOptimizer (Nested Class):

- This class represents a flight path optimizer specific to the BushfireRiskAnalyzer class. It is responsible for building the graph based on UAV data and optimizing flight paths.

- The class provides two methods:

- buildGraph(String filePath): Reads the UAV data file specified by the file path, extracts the location and data attributes (temperature, humidity, and wind speed), checks for high-risk conditions, and populates the graph and highRiskLocations.

- optimizeFlightPath(String source, Set<String> highRiskLocations): Placeholder method that returns an empty list. It represents the optimization of flight paths, which is not implemented in the provided code.

Testing methodology

(Equivalence Partitioning)

GraphOperations

Categories	**Test Data**	**Output**
Insert a location	W	Location inserted successfully.
Delete a location	W	Location deleted successfully.
Search for a location	H	Location found in the graph. Neighbors: D F G

TestHashTableImplementation

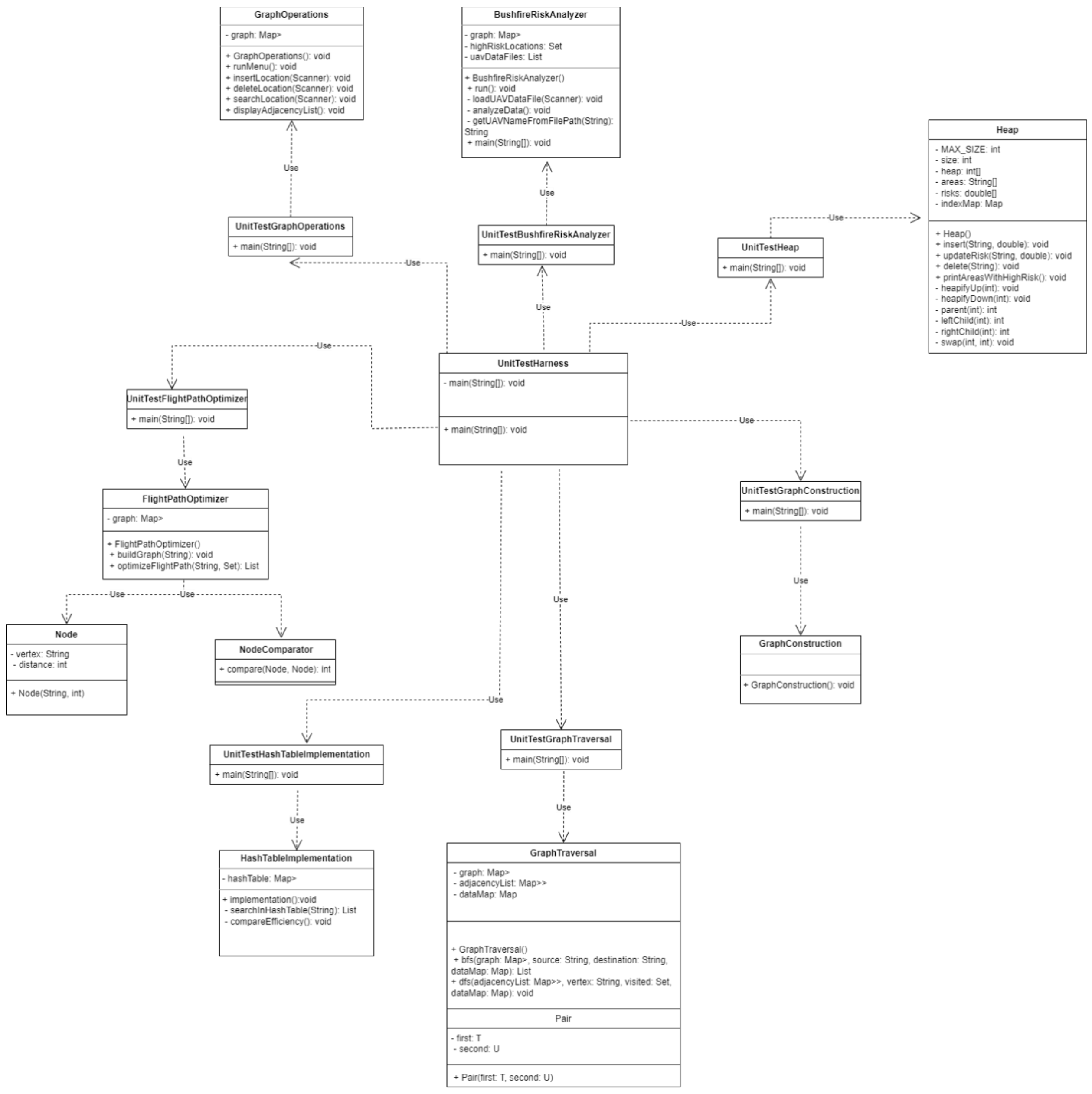
Categories	Test Data	Output
Searching for data at location	D	Data found: [2045.0, 30.0, 80.0] Comparing Efficiency: Data found using list/array: [2045.0, 30.0, 80.0] Data found using hash table: [2045.0, 30.0, 80.0] Time taken using list/array: 300400 nanoseconds Time taken using hash table: 1128400 nanoseconds
Searching for data at location	1	Data not found for location 1

		Comparing Efficiency: Data not found using hash table Time taken using list/array: 19200 nanoseconds Time taken using hash table: 205500 nanoseconds

BushfireRiskAnalyzer

Categories	**Test Data**	**Output**
Load UAV Data File	UAVdata.txt	UAV data file loaded successfully.
Analyze Data	UAVdata.txt	UAV Data File: UAVdata.txt UAV Name: UAVdata Areas of High Risk: [D, F, H, J] Itinerary: []

UML Diagram



Conclusion and Future Work:

In conclusion the program ran successfully without any errors.

As future works if I had more time I would make the code more user friendly and more efficient.