DSA ASSIGNMENT REPORT

UAV SIMULATOR

Dilhan Rubera

21026003

Content

Program Design.

UAVSimulator is a program that uses UAVs to monitor locations and determine if a location is at high risk of a bush fire. A UAV can traverse through all the locations in an area and extract data such as temperature, humidity and wind speed. Using this data gathered it can calculate a risk value and determine if an area is high risk or not. It can also traverse two locations using a start location and an end location. It does so by considering distance, thus the travel path will be the shortest path between the two locations. High risk locations may or may not be in this travel path.

In order to implement this program the following classes were used, UAVSimulator, DSAGraph, DSAGraphVertex ,DSALinkedList, DSAHeap ,DSAHeapEntry, DSAHashtable, DSAHashEntry, Pair and FileIO.

Class Descriptions.

- DSAGraph class and DSAGraphVertex class

This code is based on the Practical 6 submission.The DSAGraph class contains a list a vertices which will be used to create the graph. Methods to add, remove and search for vertexes are present. The graph traversal methods such as Depth first search, breadth first search and dijkastras algorithm are implemented in this class.

The DSAGraphVertex class will be used to create vertex's which contain a label, a DSALinked list for its adjacent vertexes, a boolean , temperature value, humidty value and wind speed value.

- DSALinkedList

This code is based on the Practical 4 submission. The DSALinkedList class is an implementation of a doubly linked double ended linked list. Methods to insert, remove, search ,traverse the list and display values are present in this class. DSAListNode is a private inner class of DSALinkedList.

- DSAHeap and DSAHeapEntry

This code is based on the Practical 8 submission. The DSAHeap class is an implementation of a max heap. Methods to add, remove, sort, extract the minimum value and display the heap are present.

- DSAHashTable and DSAHashEntry

This code is based on the practical 7 submisson. Methods to put, get, remove entries, resize and display the hash table code is implemented in DSAHashTable. Two hashing methods are implemented.

- Pair

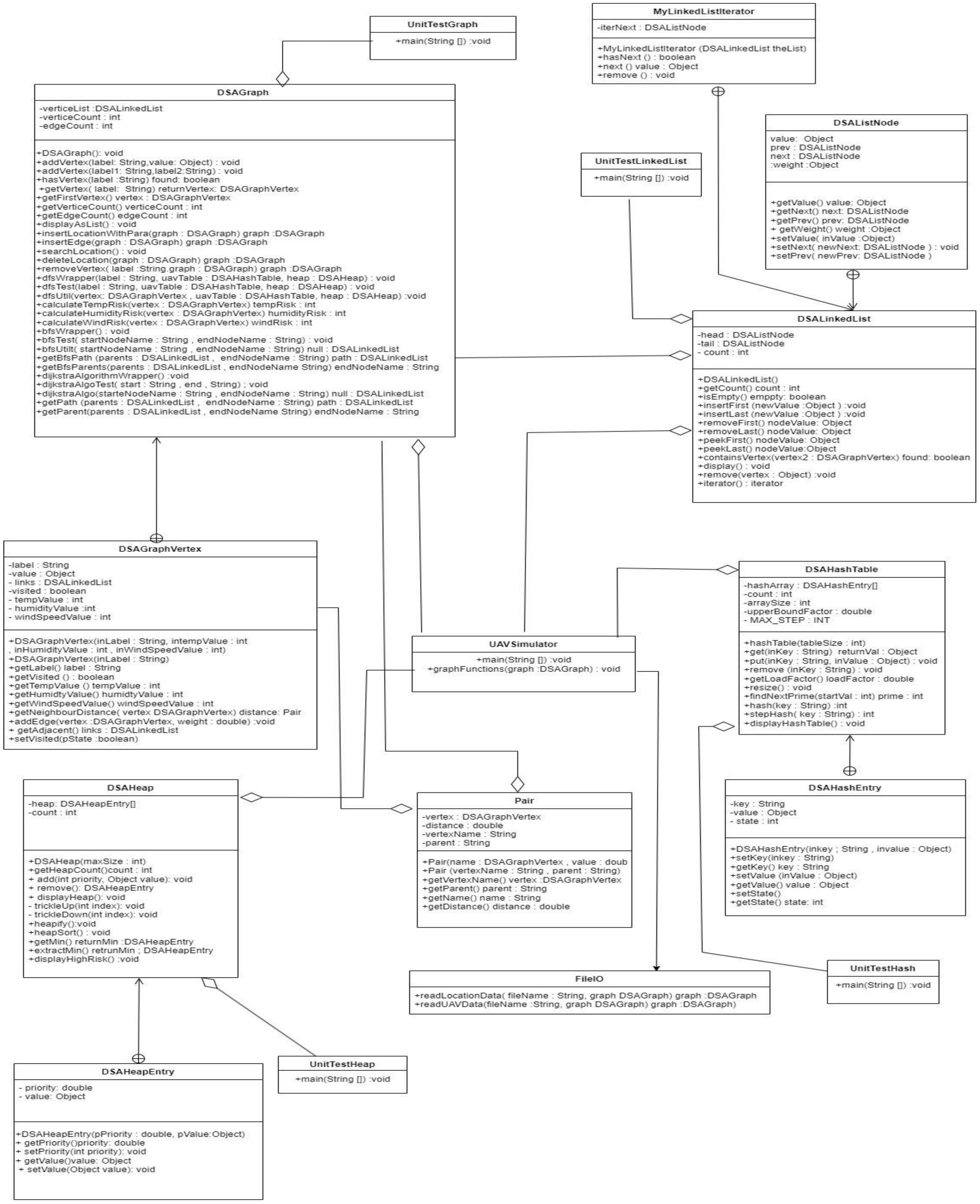
This code is used when need to pair two values together. There are two constructors implemented. The first constructor " public Pair(DSAGraphVertex name, double value) {}" will create a Pair instance which can hold a DSAGraphVertex object and a double value. This is used to store a vertex and the distance from to it from the source vertex.

The second constructor" public Pair(String vertexName, String parent) " is used to store a vertex's name and its parent. This is used in Breadth First Search and Dijkstra's Algorithm.

- UAVSimulator Class

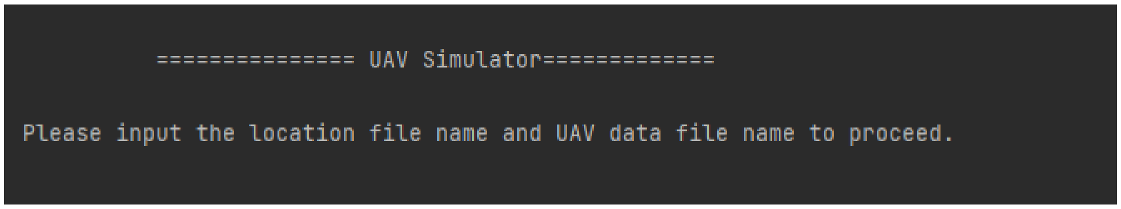
This class contains the main function which will be used to run the program.

UML Diagram.


UnitTestGraph
+main(String []) :void
DSAGraph
-verticeList :DSALinkedList
-verticeCount : int
-edgeCount : int
+DSAGraph(): void
+addVertex(label: String,value: Object) : void
+addVertex(label1: String,label2:String) : void
+hasVertex(label :String) found: boolean
+getVertex(label: String) returnVertex: DSAGraphVertex
+getFirstVertex() vertex : DSAGraphVertex
+getVerticeCount() verticeCount : int
+getEdgeCount() edgeCount : int
+displayAsList() : void
+insertLocationWithPara(graph : DSAGraph) graph :DSAGraph
+insertEdge(graph : DSAGraph) graph :DSAGraph
+searchLocation() : void
+deleteLocation(graph : DSAGraph) graph :DSAGraph
+removeVertex(label :String.graph : DSAGraph) graph :DSAGraph
+dfsWrapper(label : String, uavTable : DSAHashTable, heap : DSAHeap) : void
+dfsTest(label : String, uavTable : DSAHashTable, heap : DSAHeap) : void
+dfsUtil(vertex: DSAGraphVertex , uavTable : DSAHashTable, heap : DSAHeap) :void
+calculateTempRisk(vertex : DSAGraphVertex) tempRisk : int
+calculateHumidityRisk(vertex : DSAGraphVertex) humidityRisk : int
+calculateWindRisk(vertex : DSAGraphVertex) windRisk : int
+bfsWrapper() : void
+bfsTest(startNodeName : String , endNodeName : String) : void
+bfsUtilt(startNodeName : String , endNodeName : String) null : DSALinkedList
+getBfsPath (parents : DSALinkedList , endNodeName : String) path : DSALinkedList
+getBfsParents(parents : DSALinkedList , endNodeName String) endNodeName : String
+dijkstraAlgorithmWrapper() :void
+dijkstraAlgoTest(start : String , end , String) ; void
+dijkstraAlgo(starteNodeName : String , endNodeName : String) null : DSALinkedList
+getPath (parents : DSALinkedList , endNodeName : String) path : DSALinkedList
+getParent(parents : DSALinkedList , endNodeName String) endNodeName : String
MyLinkedListIterator
-iterNext : DSAListNode
+MyLinkedListIterator (DSALinkedList theList)
+hasNext () : boolean
+next () value : Object
+remove () : void
DSAListNode
value: Object
prev : DSAListNode
next : DSAListNode
:weight :Object
+getValue() value: Object
+getNext() next: DSAListNode
+getPrev() prev: DSAListNode
+ getWeight() weight :Object
+setValue(inValue :Object)
+setNext(newNext: DSAListNode) : void
+setPrev(newPrev: DSAListNode)
UnitTestLinkedList
+main(String []) :void
DSALinkedList
-head : DSAListNode
-tail : DSAListNode
- count : int
+DSALinkedList()
+getCount() count : int
+isEmpty() emppty: boolean
+insertFirst (newValue :Object) :void
+insertLast (newValue :Object) :void
+removeFirst() nodeValue: Object
+removeLast() nodeValue: Object
+peekFirst() nodeValue: Object
+peekLast() nodeValue:Object
+containsVertex(vertex2 : DSAGraphVertex) found: boolean
+display() : void
+remove(vertex : Object) :void
+iterator() : iterator
DSAGraphVertex
-label : String
-value : Object
- links : DSALinkedList
-visited : boolean
- tempValue : int
- humidityValue :int
- windSpeedValue : int
+DSAGraphVertex(inLabel : String, intempValue : int
, inHumidityValue : int , inWindSpeedValue : int)
+DSAGraphVertex(inLabel : String)
+getLabel() label : String
+getVisited () : boolean
+getTempValue () tempValue : int
+getHumidtyValue() humidtyValue : int
+getWindSpeedValue() windSpeedValue : int
+getNeighbourDistance(vertex DSAGraphVertex) distance: Pair
+addEdge(vertex :DSAGraphVertex, weight : double) :void
+ getAdjacent() links : DSALinkedList
+setVisited(pState :boolean)
UAVSimulator
+main(String []) :void
+graphFunctions(graph :DSAGraph) : void
DSAHashTable
-hashArray : DSAHashEntry[]
-count : int
-arraySize : int
-upperBoundFactor : double
- MAX_STEP : INT
+hashTable(tableSize : int)
+get(inKey : String) returnVal : Object
+put(inKey : String, inValue : Object) : void
+remove (inKey : String) : void
+getLoadFactor() loadFactor : double
+resize() : void
+findNextPrime(startVal : int) prime : int
+hash(key : String) :int
+stepHash(key : String) : int
+displayHashTable() : void
DSAHeap
-heap: DSAHeapEntry[]
-count : int
+DSAHeap(maxSize : int)
+getHeapCount()count : int
+ add(int priority, Object value): void
+ remove(): DSAHeapEntry
+ displayHeap(): void
- trickleUp(int index): void
- trickleDown(int index): void
+heapify():void
+heapSort() : void
+getMin() returnMin :DSAHeapEntry
+extractMin() retrunMin ; DSAHeapEntry
+displayHighRisk() :void
Pair
-vertex : DSAGraphVertex
-distance : double
-vertexName : String
-parent : String
+Pair(name : DSAGraphVertex , value : doub
+Pair (vertexName : String , parent : String)
+getVertexName() vertex :DSAGraphVertex
+getParent() parent : String
+getName() name : String
+getDistance() distance : double
DSAHashEntry
-key : String
-value : Object
- state : int
+DSAHashEntry(inkey ; String , invalue : Object)
+setKey(inkey : String)
+getKey() key : String
+setValue (inValue : Object)
+getValue() value : Object
+setState()
+getState() state: int
UnitTestHash
+main(String []) :void
FileIO
+readLocationData(fileName : String, graph DSAGraph) graph :DSAGraph
+readUAVData(fileName :String, graph DSAGraph) graph :DSAGraph)
DSAHeapEntry
- priority: double
- value: Object
+DSAHeapEntry(pPriority : double, pValue:Object)
+ getPriority()priority: double
+ setPriority(int priority): void
+ getValue()value: Object
+ setValue(Object value): void
UnitTestHeap
+main(String []) :void

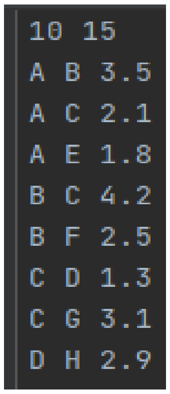
UAVSimulator’s Functionality.

- When a user runs the program from the UAVSimulator class he will first see the following message.



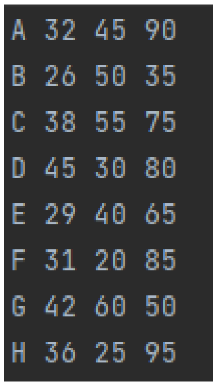
```
          =============== UAV Simulator=============

Please input the location file name and UAV data file name to proceed.
```

- The user must enter a location file name (File with the edges) first. The given below file for reference.



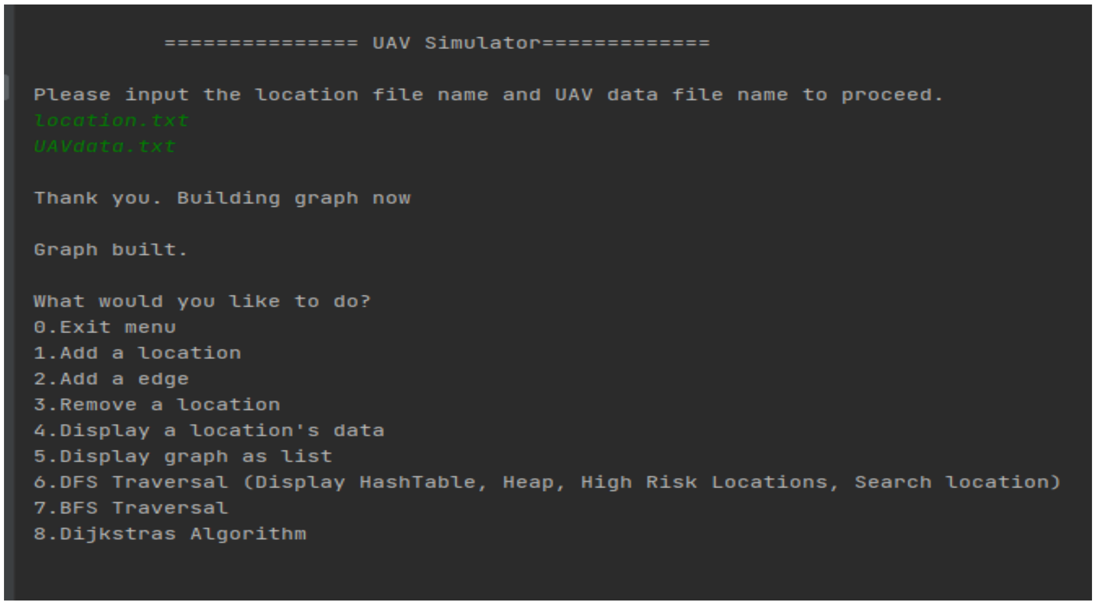
```
10 15
A B 3.5
A C 2.1
A E 1.8
B C 4.2
B F 2.5
C D 1.3
C G 3.1
D H 2.9
```

- The file name with the UAV data must be entered second. The given below file for reference.



```
A 32 45 90
B 26 50 35
C 38 55 75
D 45 30 80
E 29 40 65
F 31 20 85
G 42 60 50
H 36 25 95
```

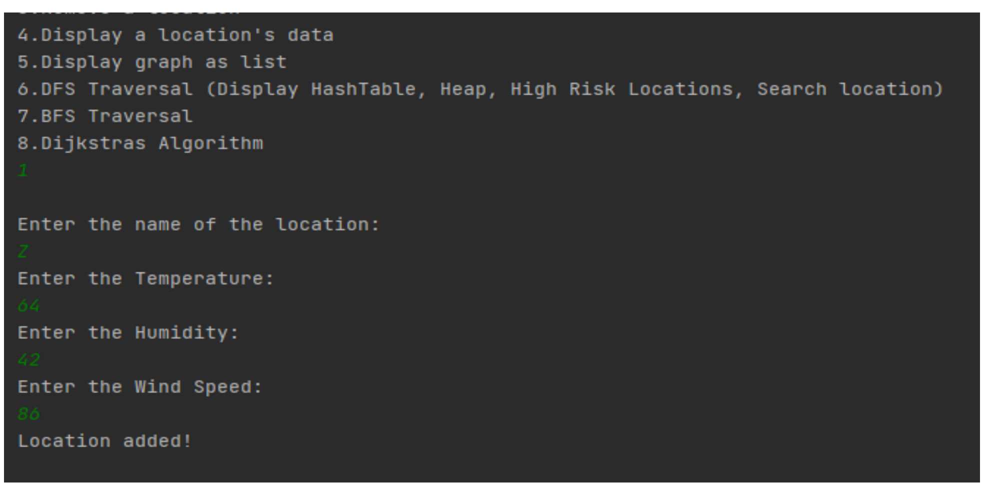
- Once the file name has been entered the user will be given 9 functionalities to choose from.



```
          =============== UAV Simulator=============

Please input the location file name and UAV data file name to proceed.
location.txt
UAVdata.txt

Thank you. Building graph now

Graph built.

What would you like to do?
0.Exit menu
1.Add a location
2.Add a edge
3.Remove a location
4.Display a location's data
5.Display graph as list
6.DFS Traversal (Display HashTable, Heap, High Risk Locations, Search location)
7.BFS Traversal
8.Dijkstras Algorithm
```

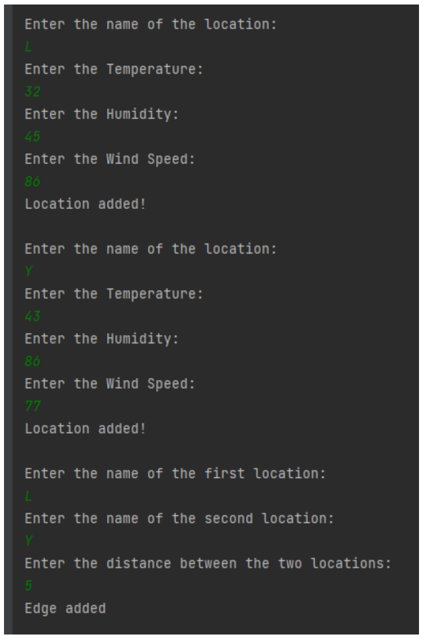
0) A user can exit the simulation by inputting 0.

1) A user can insert a location as follows by inputting 1 .



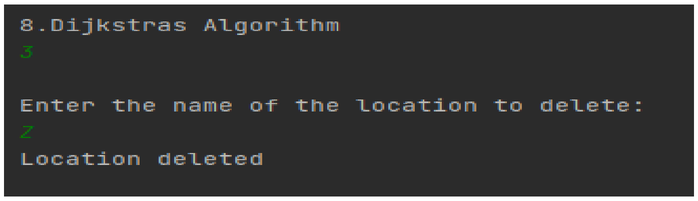
```
4.Display a location's data
5.Display graph as list
6.DFS Traversal (Display HashTable, Heap, High Risk Locations, Search location)
7.BFS Traversal
8.Dijkstras Algorithm
1

Enter the name of the location:
Z
Enter the Temperature:
64
Enter the Humidity:
42
Enter the Wind Speed:
86
Location added!
```

2) User can insert an edge as follows by inputting 2. Two vertexes needed to be added first. Once they are added enter the first location, the second and the distance between them.



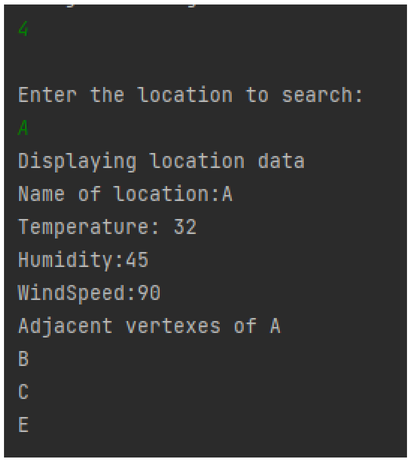
```
Enter the name of the location:
L
Enter the Temperature:
32
Enter the Humidity:
45
Enter the Wind Speed:
86
Location added!

Enter the name of the location:
Y
Enter the Temperature:
43
Enter the Humidity:
86
Enter the Wind Speed:
77
Location added!

Enter the name of the first location:
L
Enter the name of the second location:
Y
Enter the distance between the two locations:
5
Edge added
```

3) User can remove a location by inputting 3. Insert the name of the location to be deleted.



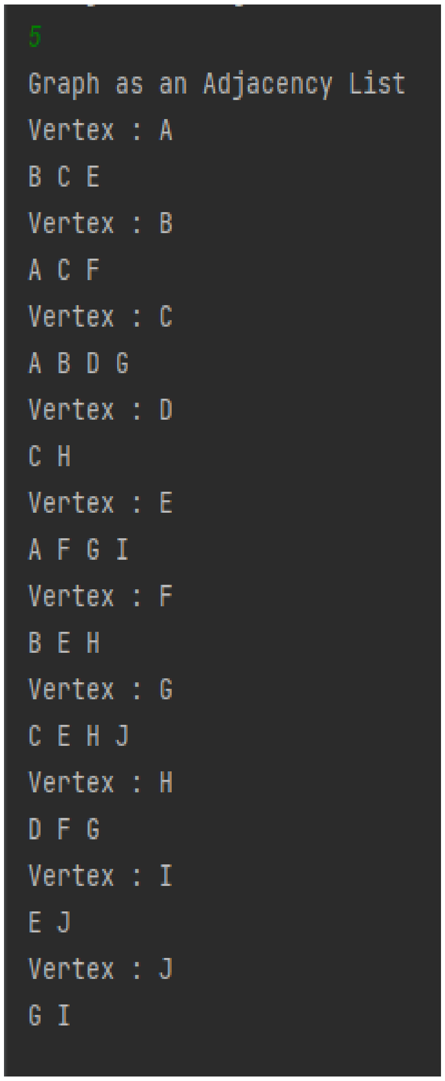
```
8.Dijkstras Algorithm
3

Enter the name of the location to delete:
Z
Location deleted
```

4) User can display a display a location's data by inputting 4. Insert the name of location to be displayed.



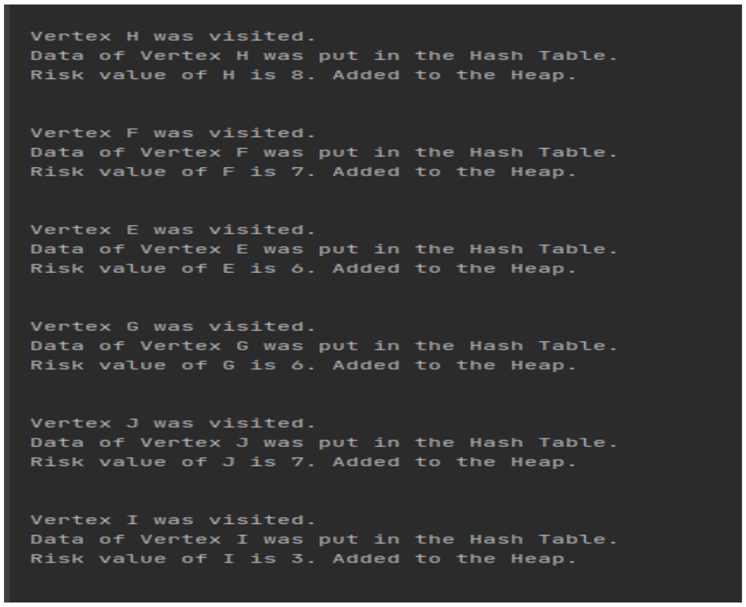
```
4

Enter the location to search:
A
Displaying location data
Name of location:A
Temperature: 32
Humidity:45
WindSpeed:90
Adjacent vertexes of A
B
C
E
```

5) User can display graph as an adjacency list by inserting 5.



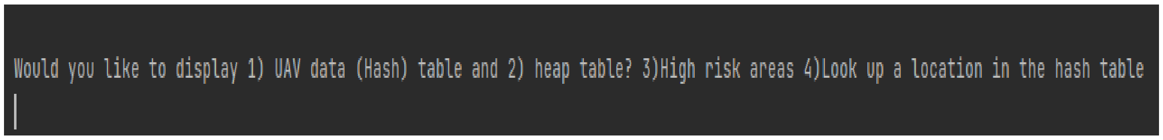
```
5
Graph as an Adjacency List
Vertex : A
B C E
Vertex : B
A C F
Vertex : C
A B D G
Vertex : D
C H
Vertex : E
A F G I
Vertex : F
B E H
Vertex : G
C E H J
Vertex : H
D F G
Vertex : I
E J
Vertex : J
G I
```

6) User can carry out Depth First Search by inputting 6.



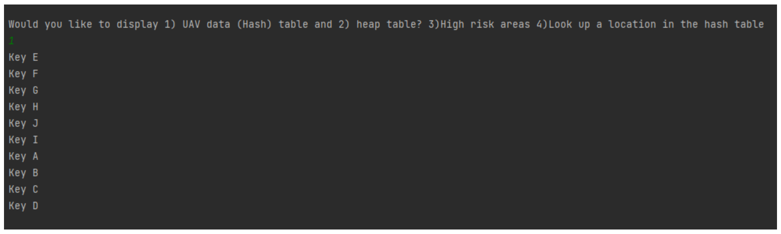
```
Vertex H was visited.
Data of Vertex H was put in the Hash Table.
Risk value of H is 8. Added to the Heap.


Vertex F was visited.
Data of Vertex F was put in the Hash Table.
Risk value of F is 7. Added to the Heap.


Vertex E was visited.
Data of Vertex E was put in the Hash Table.
Risk value of E is 6. Added to the Heap.


Vertex G was visited.
Data of Vertex G was put in the Hash Table.
Risk value of G is 6. Added to the Heap.


Vertex J was visited.
Data of Vertex J was put in the Hash Table.
Risk value of J is 7. Added to the Heap.


Vertex I was visited.
Data of Vertex I was put in the Hash Table.
Risk value of I is 3. Added to the Heap.
```

Once all vertices are visited the user will see this message. This is the DFS sub menu.



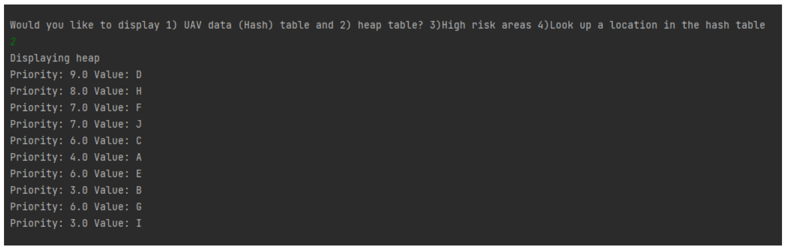
```
Would you like to display 1) UAV data (Hash) table and 2) heap table? 3)High risk areas 4)Look up a location in the hash table
|
```

i) By inputting 1 a user can display the hash table entries.



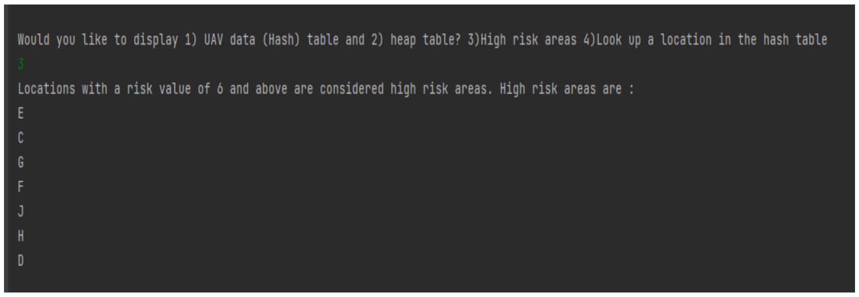
```
Would you like to display 1) UAV data (Hash) table and 2) heap table? 3)High risk areas 4)Look up a location in the hash table
1
Key E
Key F
Key G
Key H
Key J
Key I
Key A
Key B
Key C
Key D
```

ii) By inputting 2 a user can display the heap.



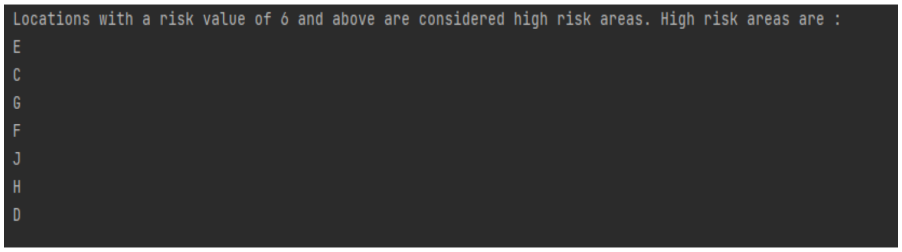
```
Would you like to display 1) UAV data (Hash) table and 2) heap table? 3)High risk areas 4)Look up a location in the hash table
2
Displaying heap
Priority: 9.0 Value: D
Priority: 8.0 Value: H
Priority: 7.0 Value: F
Priority: 7.0 Value: J
Priority: 6.0 Value: C
Priority: 4.0 Value: A
Priority: 6.0 Value: E
Priority: 3.0 Value: B
Priority: 6.0 Value: G
Priority: 3.0 Value: I
```

iii) By inputting 3 a user can display the high risk areas



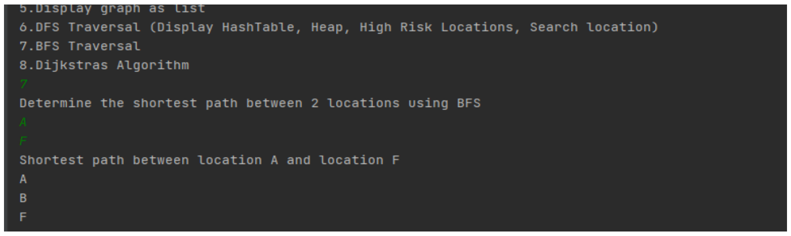
```
Would you like to display 1) UAV data (Hash) table and 2) heap table? 3)High risk areas 4)Look up a location in the hash table
3
Locations with a risk value of 6 and above are considered high risk areas. High risk areas are :
E
C
G
F
J
H
D
```

iv) By inputting 4 a user can display a location from the hash table



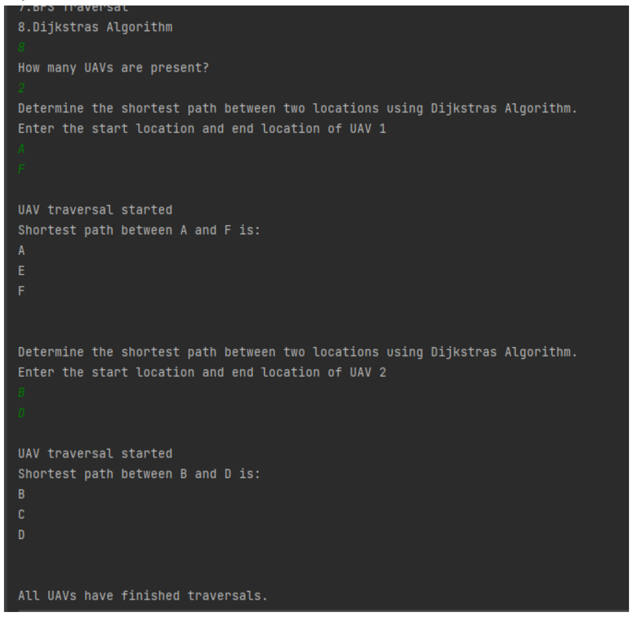
```
Locations with a risk value of 6 and above are considered high risk areas. High risk areas are :
E
C
G
F
J
H
D
```

7) By inputting 7 a user can find the shortest path between the two locations using Breadth first search. This implementation of BFS does not consider the distance between the two locations.



```
5.Display graph as list
6.DFS Traversal (Display HashTable, Heap, High Risk Locations, Search location)
7.BFS Traversal
8.Dijkstras Algorithm
7
Determine the shortest path between 2 locations using BFS
A
F
Shortest path between location A and location F
A
B
F
```

8) By inputting 8 a user can find the shortest path between the two locations using Dijkstra's Algorithm. This implementation does take into consideration the distance between the two locations. The user is given the option to determine the number of UAVs that will be traversing.



```
7.BFS Traversal
8.Dijkstras Algorithm
8
How many UAVs are present?
2
Determine the shortest path between two locations using Dijkstras Algorithm.
Enter the start location and end location of UAV 1
A
F

UAV traversal started
Shortest path between A and F is:
A
E
F


Determine the shortest path between two locations using Dijkstras Algorithm.
Enter the start location and end location of UAV 2
B
D

UAV traversal started
Shortest path between B and D is:
B
C
D


All UAVs have finished traversals.
```

Data Structures implementation and justification.

DSAGraph class and DSAGraphVertex class

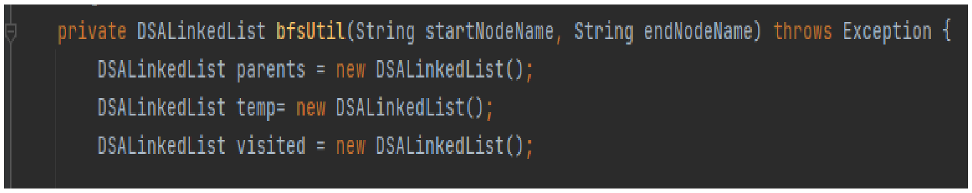
To implement the UAVSimulator we need to implement a graph that would have all the locations and the distances between them. To implement this DSAGraph class and DSAGraphVertex class was used.
An instance of the DSAGraph class was used as a graph to map out the area with locations that the UAVs traverse. A DSAGraphVertex class instance was used as a single location that the UAVs could traverse to. Therefore the graph consists of multiple location vertexes, hence why using these classes was ideal for this program.

DSALinkedList

Instances of DSALinkedList are used in multiple scenarios throughout the program. The DSALinkedList class creates objects that are Doubly Linked Double Ended linked lists.

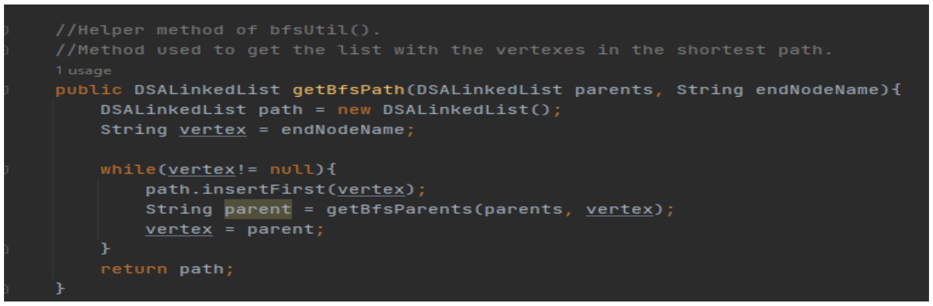
In the DSAGraph class all the vertex locations are inserted in a DSALinked list instance called verticeList.
In every DSAGraphVertex object a DSALinkedList instance is present to store the adjacent vertex name and distance to it from the source vertex.

In the Breadth First Search implementation (bsfUtil()), instances of DSALinkedList are used several times. Lists are used to store the parent vertexes of a vertex, to store the temporary vertexes (the vertexes that are to be visited) and to store the vertexes that have been visited.



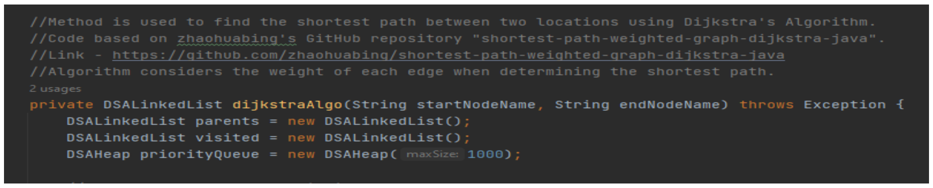
```
private DSALinkedList bfsUtil(String startNodeName, String endNodeName) throws Exception {
    DSALinkedList parents = new DSALinkedList();
    DSALinkedList temp= new DSALinkedList();
    DSALinkedList visited = new DSALinkedList();
```

In order to print out the shortest path , the vertexes of the shortest path are stored in a linked list.



```
//Helper method of bfsUtil().
//Method used to get the list with the vertexes in the shortest path.
1 usage
public DSALinkedList getBfsPath(DSALinkedList parents, String endNodeName){
    DSALinkedList path = new DSALinkedList();
    String vertex = endNodeName;

    while(vertex!= null){
        path.insertFirst(vertex);
        String parent = getBfsParents(parents, vertex);
        vertex = parent;
    }
    return path;
}
```

In the implementation of Dijkstra’s Algorithm instances of DSALinkedLists are used to store the parent vertexes and the visited vertexes of the graph. To print the shortest path, the vertexes of the shortest path are stored in a list.



```java
//Method is used to find the shortest path between two locations using Dijkstra's Algorithm.
//Code based on zhaohuabing's GitHub repository "shortest-path-weighted-graph-dijkstra-java".
//Link - https://github.com/zhaohuabing/shortest-path-weighted-graph-dijkstra-java
//Algorithm considers the weight of each edge when determining the shortest path.
2 usages
private DSALinkedList dijkstraAlgo(String startNodeName, String endNodeName) throws Exception {
    DSALinkedList parents = new DSALinkedList();
    DSALinkedList visited = new DSALinkedList();
    DSAHeap priorityQueue = new DSAHeap( maxSize: 1000);
```

DSALinkedList instances were ideal to implement the above scenarios due to a linked list having the ability to dynamically allocate memory. As the program needs to be scalable and we are unaware of the number of vertexes at the compile time of the program, we need a data structure that can dynamically allocate memory to grow or shrink with values. An array is not ideal for this as the size of the array needs to be know at the compile time of the program. Values can be easily inserted and removed from the list as well.

The drawback of using a linked list over an array is that the access time in an array is way faster due to the direct access to an element.

DSAHashTable and DSAHashEntry

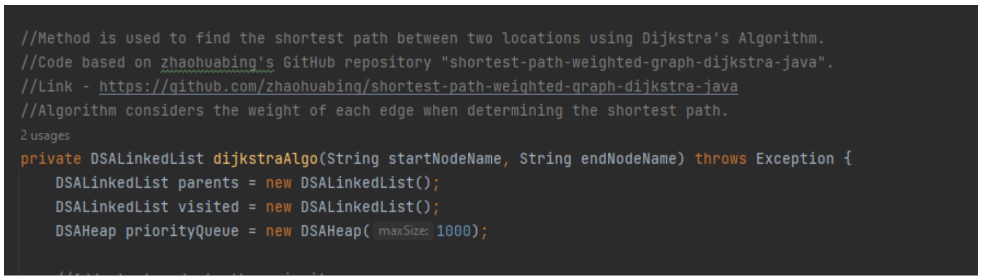
In this program a DSAHashTable is used to store the vertexes with its relevant value. A DSAHeapEntry instance will consist of the vertex name as the key which will be hashed, and the vertex itself will be the object value. The hash table is also used to locate a location and display its value in the menu.
The hashtable is ideal in order to retrieve a location's data due to its fast access time to a entry object. Thus, its more suitable to store and display values when compared to an instance of a DSALinkedList.

DSAHeap and DSAHeapEntry

Two instances of DSAHeap are used in this program.

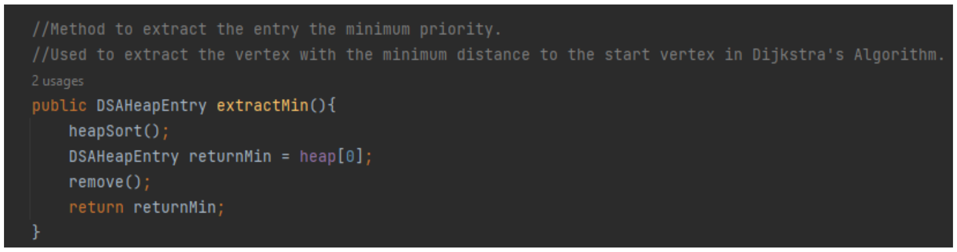
Once the UAVs conduct a depth first search on the graph and visit every vertex, the UAV calculates a risk value and stores the risk value and vertex in a heap. In this scenario the risk value is considered the priority and the vertex is considered the value. The vertexes with the highest risk values will be at the top of the heap. The program requires us to display the locations with the highest risk values. A heap is ideal for this as it stores values in priority order with the corresponding value.

An instance of DSAHeap is also used when implementing Dijkstra's Algorithm. The GitHub code that the algorithm was based on required a priority queue. Thus to implement the priority queue a heap was used. When conducting Dijkastras algorithm a we need a way to retrieve an adjacent vertex which is closest to the vertex. Thus in order to implement this we need a data structure that can be used to easily retrieve an object with a minimum distance (minimum priority). Therefore, a heap is ideal in this implementation.



```java
//Method is used to find the shortest path between two locations using Dijkstra's Algorithm.
//Code based on zhaohuabing's GitHub repository "shortest-path-weighted-graph-dijkstra-java".
//Link - https://github.com/zhaohuabing/shortest-path-weighted-graph-dijkstra-java
//Algorithm considers the weight of each edge when determining the shortest path.
2 usages
private DSALinkedList dijkstraAlgo(String startNodeName, String endNodeName) throws Exception {
    DSALinkedList parents = new DSALinkedList();
    DSALinkedList visited = new DSALinkedList();
    DSAHeap priorityQueue = new DSAHeap( maxSize: 1000);
```

A method called extractMin() was created in the DSAHeap class. This method sorts the heap, and then removes and returns the value at the top of the heap. Since the heap is sorted the value with smallest priority(minimum distance) is removed and returned. Thus the heap (priority queue) ensures that the adjacent vertex with the smallest distance is always extracted from the heap.



```
//Method to extract the entry the minimum priority.
//Used to extract the vertex with the minimum distance to the start vertex in Dijkstra's Algorithm.
2 usages
public DSAHeapEntry extractMin(){
    heapSort();
    DSAHeapEntry returnMin = heap[0];
    remove();
    return returnMin;
}
```

Task 2

The breadth first search implementation used in the program was based on zhaohuabing's GitHub repository " shortest path-unweighted-graph-bsf-java ", link - https://github.com/zhaohuabing/shortest-path-unweighted-graph-bsf-java .

Please note that this implementation does not consider the weight (the distance) between two vertexes. This is because BFS can not be implemented in weighted graphs. Therefore, this implementation finds the shortest path within two locations without the distance constraint. To find the shortest path while considering the weight, Dijkstra's Algorithm is used which will be discussed in the task 6 section.

Task 4

When the UAVs visit every vertex while conducting Depth First Search, the data of each vertex is retrieved. The retrieved data is then stored in the hashtable along with the vertex name. In the menu the user is given the option to conduct Depth First Search.

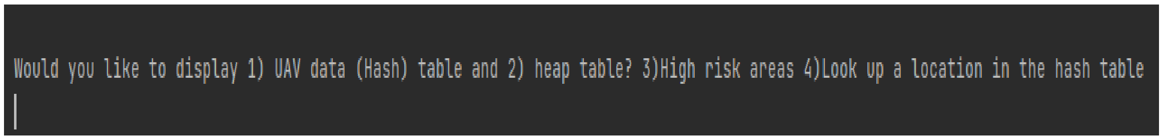
Once the UAVs have finished traversal the user is given the option to display the keys (the vertex names) in the hashtable or to lookup any location in the hashtable and display its data.

When retrieving stored data, a heap is more efficient in comparison to a linked list. The reason being that once the key is given to the hash function hashes the key and retrieves the value from the hash table. Thus giving the hashtable an access time varying from O(1) to O(N) but its highly likely that the access time is at the lower end (O(1)). A linked list would require a traversal through it in order to find the required value. Thus, the access time to find a value will be always be O(N). Therefore, we can come to the conclusion that using a hashtable to retrieve stored data is more efficient compared to a linked list.

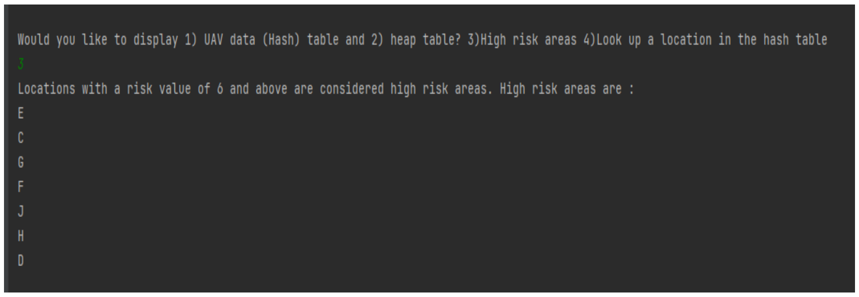
Task 6 – Itinerary

In order to optimize the UAVs flight paths between two locations ,the shortest path between them must be found. To implement this Dijkstra's Algorithm was used. The code implemented was based on zhaohuabing's GitHub repository "[shortest-path-weighted-graph-dijkstra-java](https://github.com/zhaohuabing/shortest-path-weighted-graph-dijkstra-java)" , link - https://github.com/zhaohuabing/shortest-path-weighted-graph-dijkstra-java.Dijkkstra's algorithm can be used to find the shortest path between two locations while considering the distance between the two locations, hence the reason for its implementation in this program.

The algorithm can be used to find the shortest path between two high risk locations given that the UAVs already found the high risk areas while conducting Depth First Search (High risk areas are found during DFS, they can be displayed from DFS sub menu).

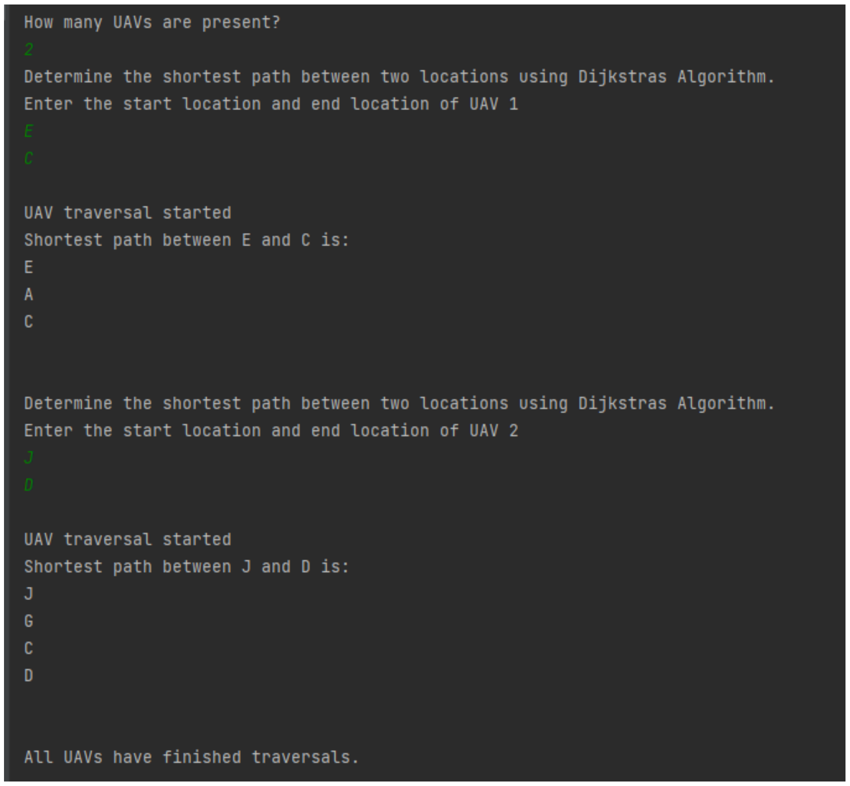


```
Would you like to display 1) UAV data (Hash) table and 2) heap table? 3)High risk areas 4)Look up a location in the hash table
```



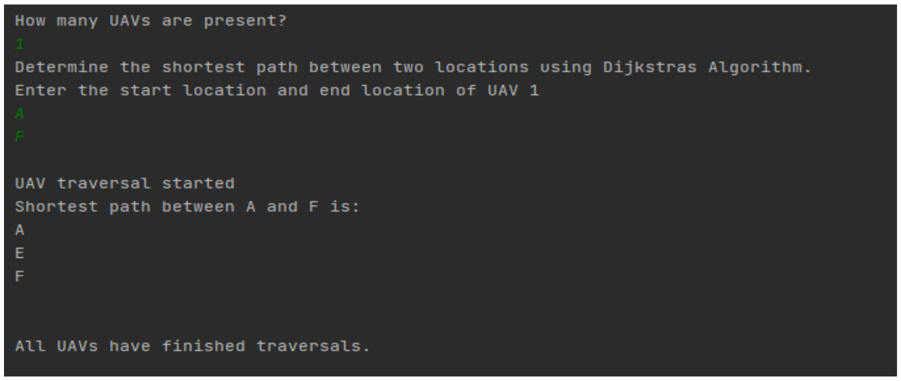
```
Would you like to display 1) UAV data (Hash) table and 2) heap table? 3)High risk areas 4)Look up a location in the hash table
3
Locations with a risk value of 6 and above are considered high risk areas. High risk areas are :
E
C
G
F
J
H
D
```

In order to make the program scalable the user will be able to choose the number of UAVs traversing. The user can enter the start and end locations for each UAV. An example of the shortest path between two high risk locations using two UAVs :



```
How many UAVs are present?
2
Determine the shortest path between two locations using Dijkstras Algorithm.
Enter the start location and end location of UAV 1
E
C

UAV traversal started
Shortest path between E and C is:
E
A
C


Determine the shortest path between two locations using Dijkstras Algorithm.
Enter the start location and end location of UAV 2
J
D

UAV traversal started
Shortest path between J and D is:
J
G
C
D


All UAVs have finished traversals.
```

The algorithm can be used to find the shortest path between any two locations as well. Although high risk areas may or may not be present in this shortest path. An example of the shortest path between A and F. The shortest path is A , E, F , thus the shortest path includes one high risk location (E).



```
How many UAVs are present?
1
Determine the shortest path between two locations using Dijkstras Algorithm.
Enter the start location and end location of UAV 1
A
F

UAV traversal started
Shortest path between A and F is:
A
E
F


All UAVs have finished traversals.
```

Testing Methodology.

For DSAGraph class , DSALinkedList class , DSAHeap class and DSAHash class test harness have been made. All methods used in their relevant classes have been tested using hardcoded inputs.

For DSAGraph class, the test harness GraphUnitTest contains the location.txt and UAVdata.txt files hardcoded to it. All graph functions, traversal methods can be tested out.

Conclusion and future improvements

Even though all the functionalities work, I think it would have been better if the outputs were displayed in a more user-friendly manner. Also, I believe that the menu could have been more in line with the concept of a UAV Simulator.