DSA Assignment

Jhon Gonzales

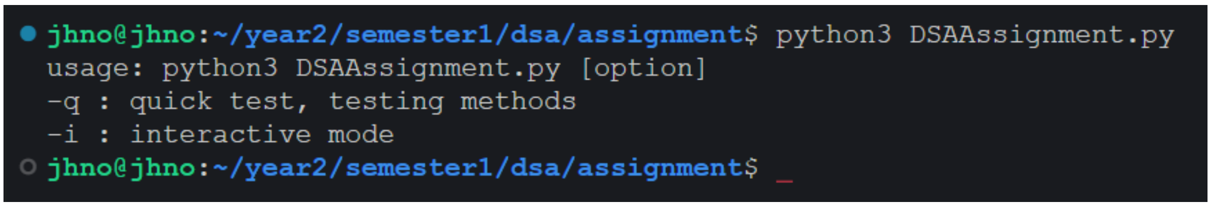
20948685

User Guide

This program runs on two main python files, scalabilityProgram.py and DSAAssignment.py. scalabilityProgram.py interacts with the python classes in the Dependencies folder directly, whereas the DSAAssignment.py program interacts with a wrapper class called DSAUav. This class wraps up all the main methods into one UAV class for user interaction. The scalability program is mostly used for editing data, creating new locations and import and export of data files. DSAAssignment.py has two main command line arguments, “-i”, “-q”, “-i” is the argument used to launch the user interface, and “-q” is used for a quick test of all methods, all outputted to the terminal. scalabilityProgram.py does not require any command line arguments, but when opened, a carat “> “is used to type in commands to edit files.

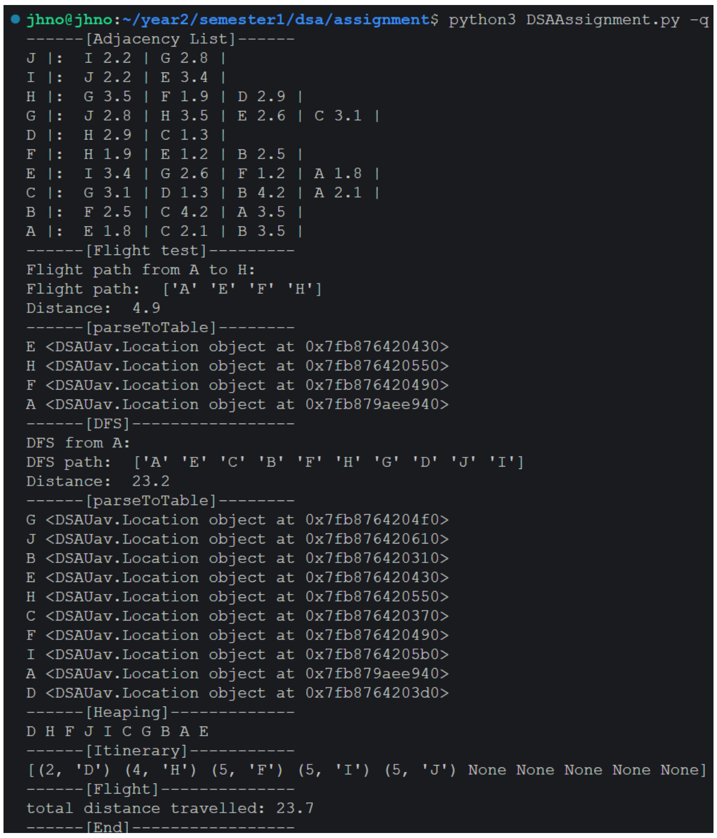
Both files are case sensitive, meaning that instead of being outputted a list of methods and a corresponding number associated with the method, a carat “> “is used to type commands to use the program.

- An example of running DSAAssignment.py without any command line arguments



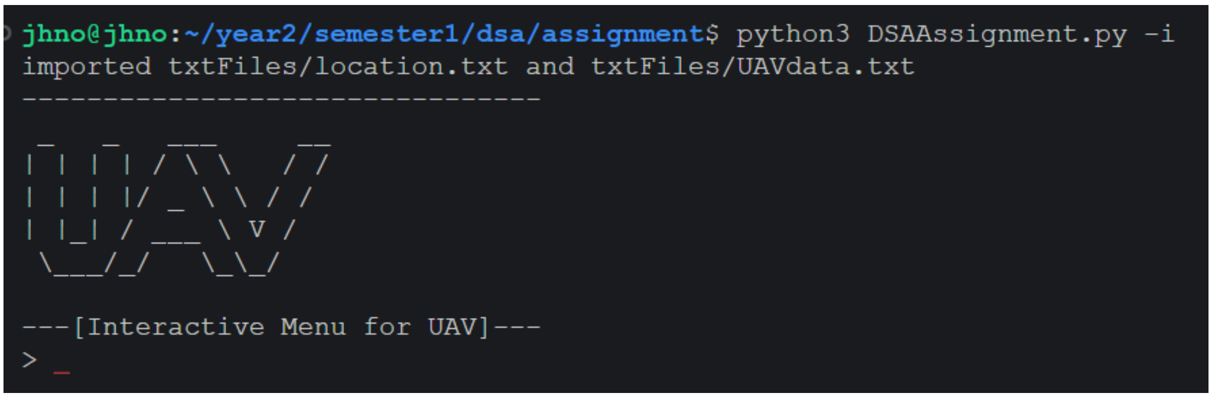
```
jhno@jhno:~/year2/semester1/dsa/assignment$ python3 DSAAssignment.py
usage: python3 DSAAssignment.py [option]
-q : quick test, testing methods
-i : interactive mode
jhno@jhno:~/year2/semester1/dsa/assignment$ _
```

- An example of running DSAAssignment.py with “-q” argument



```
jhno@jhno:~/year2/semester1/dsa/assignment$ python3 DSAAssignment.py -q
------[Adjacency List]------
J |:  I 2.2 | G 2.8 |
I |:  J 2.2 | E 3.4 |
H |:  G 3.5 | F 1.9 | D 2.9 |
G |:  J 2.8 | H 3.5 | E 2.6 | C 3.1 |
D |:  H 2.9 | C 1.3 |
F |:  H 1.9 | E 1.2 | B 2.5 |
E |:  I 3.4 | G 2.6 | F 1.2 | A 1.8 |
C |:  G 3.1 | D 1.3 | B 4.2 | A 2.1 |
B |:  F 2.5 | C 4.2 | A 3.5 |
A |:  E 1.8 | C 2.1 | B 3.5 |
------[Flight test]---------
Flight path from A to H:
Flight path:  ['A' 'E' 'F' 'H']
Distance:  4.9
------[parseToTable]--------
E <DSAUav.Location object at 0x7fb876420430>
H <DSAUav.Location object at 0x7fb876420550>
F <DSAUav.Location object at 0x7fb876420490>
A <DSAUav.Location object at 0x7fb879aee940>
------[DFS]-----------------
DFS from A:
DFS path:  ['A' 'E' 'C' 'B' 'F' 'H' 'G' 'D' 'J' 'I']
Distance:  23.2
------[parseToTable]--------
G <DSAUav.Location object at 0x7fb8764204f0>
J <DSAUav.Location object at 0x7fb876420610>
B <DSAUav.Location object at 0x7fb876420310>
E <DSAUav.Location object at 0x7fb876420430>
H <DSAUav.Location object at 0x7fb876420550>
C <DSAUav.Location object at 0x7fb876420370>
F <DSAUav.Location object at 0x7fb876420490>
I <DSAUav.Location object at 0x7fb8764205b0>
A <DSAUav.Location object at 0x7fb879aee940>
D <DSAUav.Location object at 0x7fb8764203d0>
------[Heaping]-------------
D H F J I C G B A E
------[Itinerary]-----------
[(2, 'D') (4, 'H') (5, 'F') (5, 'I') (5, 'J') None None None None None]
------[Flight]--------------
total distance travelled: 23.7
------[End]-----------------
```

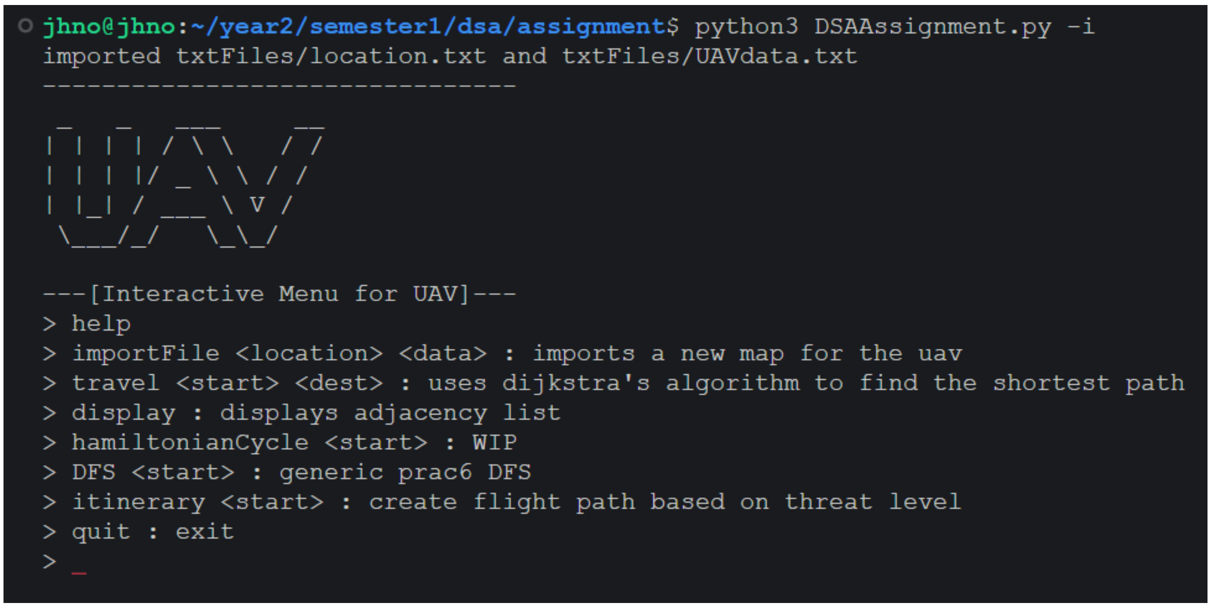
- Running DSAAssignment.py with “-i” argument



```
jhno@jhno:~/year2/semester1/dsa/assignment$ python3 DSAAssignment.py -i
imported txtFiles/location.txt and txtFiles/UAVdata.txt
--------------------------------
 _   _    ___     __
| | | |  / \ \   / /
| | | | / _ \ \ / /
| |_| |/ ___ \ V /
 \___/_/    \_\_/

---[Interactive Menu for UAV]---
> _
```

- The carat “> “allows the user to type in methods and arguments to interact with the program.

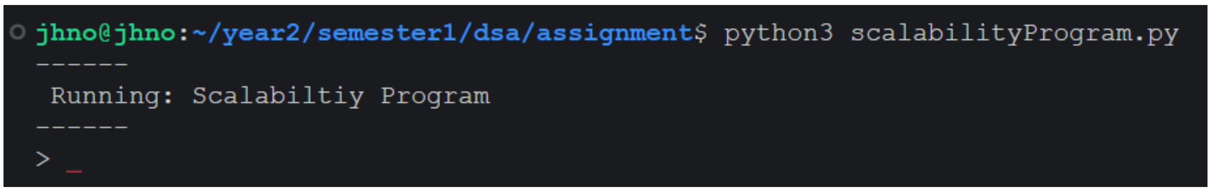
- The list of methods are outputted using the “h”, for help command



```
jhno@jhno:~/year2/semester1/dsa/assignment$ python3 DSAAssignment.py -i
imported txtFiles/location.txt and txtFiles/UAVdata.txt
--------------------------------
 _   _    ___     __
| | | |  / \ \   / /
| | | | / _ \ \ / /
| |_| |/ ___ \ V /
 \___/_/    \_\_/

---[Interactive Menu for UAV]---
> help
> importFile <location> <data> : imports a new map for the uav
> travel <start> <dest> : uses dijkstra's algorithm to find the shortest path
> display : displays adjacency list
> hamiltonianCycle <start> : WIP
> DFS <start> : generic prac6 DFS
> itinerary <start> : create flight path based on threat level
> quit : exit
> _
```

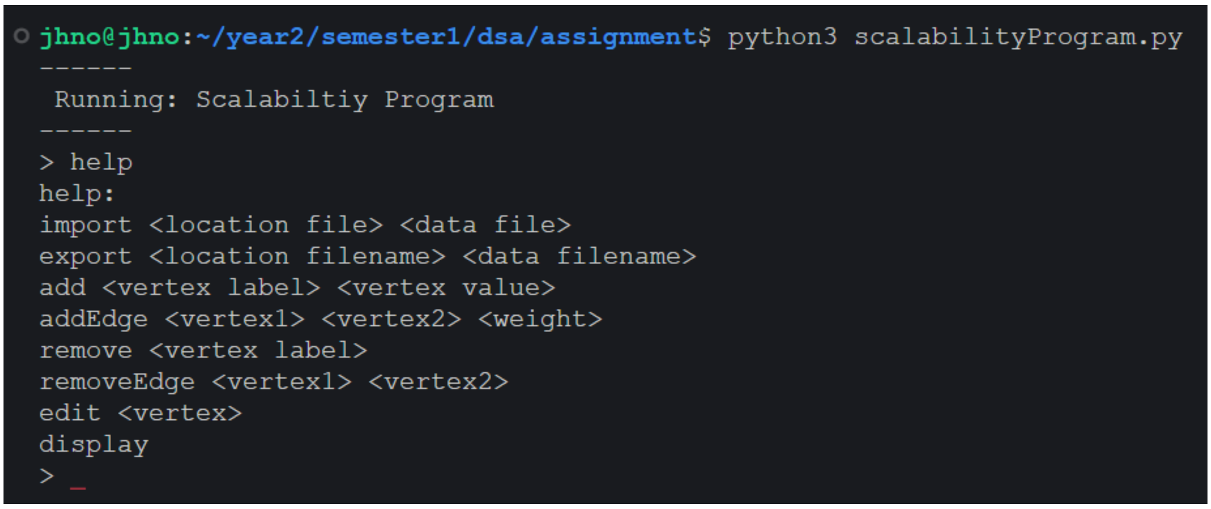
For scalabilityProgram.py:

- The main menu of ScalabilityProgram.py
- Just like DSAAssignment.py, a carat allows the user to type in commands rather than picking from an assortment of numbers.



```
jhno@jhno:~/year2/semester1/dsa/assignment$ python3 scalabilityProgram.py
------
 Running: Scalabiltiy Program
------
> _
```

- using the “help” command on the program shows the user the commands available.



```
jhno@jhno:~/year2/semester1/dsa/assignment$ python3 scalabilityProgram.py
------
 Running: Scalabiltiy Program
------
> help
help:
import <location file> <data file>
export <location filename> <data filename>
add <vertex label> <vertex value>
addEdge <vertex1> <vertex2> <weight>
remove <vertex label>
removeEdge <vertex1> <vertex2>
edit <vertex>
display
> _
```

- Command descriptions:
  - help: displays the help menu.
  - import <location file> <data file>: imports the data file to create a graph, the layout of the graph has to be the same format as locations.txt and UAVdata.txt.
  - add <vertex label> <vertex value>: creates a new vertex in the graph, whether new or imported.
  - addEdge <veretex1> <vertex2> <weight>: creates an edge of <weight> weight between two vertices, it creates an undirected edge, not a directed edge, meaning both vertices are able to traverse to each other.
  - remove <vertex label>: removes the vertex from the graph and its existing edges to other vertices
  - removeEdge <vertex1> <vertex2>: removes an edge from the two vertices.
  - edit <vertex>: leads to a vertex edit method that allows the user to edit the label, value, add and remove edges.
  - display: displays the graph as an adjacency list.

Description of Classes

In total, the UAV interacts with 5 different classes, notably:

- DSAGraph.py, from practical 6
    - DSAGraph.py is a heavily modified class file, with new methods, allowing the creation of new edges with weights, new search algorithms and better remove methods.

- DSAhash.py, from practical 7
    - DSAhash.py is slightly modified from its original format, the only difference is that it now allows the hashing of strings, rather than integers.

- DSAHeap.py, from practical 8
    - DSAHeap.py still remains the same.

- DSALinkedList.py, from practical 4
    - DSALinkedList.py now has improved removal methods, a new removeItem() method that allows the removal of any node at any location in the linked list.

- DSAUav.py, a new wrapper class combining all classes into a user interactable class.
    - This wrapper class wraps up DSAGraph's traversal methods, hashing and heaping all in one class, rather than having to be implemented every time.

Justification of Decisions:

The following data structures are re-used from Practicals 4, 6, 7, 8:

- Linked Lists, Graphs, Hash Tables, and Heaps

These are the fundamental data structures required as per specification of the tasks. There was a possibility of using queues and stacks from Practical 3, however, linked lists were an easier addition as it can perform both queues and stack behaviours through insert first/last and remove first/last. One major advantage of using linked lists was the new method of remove Item, this method allowed the ability of removing any linked list node at any location in the list, whether it was the first, last or in the middle of the list. This is helpful towards the scalability program where the user is allowed to remove any locations and edges from the graph object. The implementation of the linked lists was roughly the same as it was per Practical 4, however some methods were improved, and new methods were implemented.

One major advantage of using linked lists is the ability to store any number of objects inside, which means that the program can theoretically store as much data without running into size issues unlike arrays which have a fixed size, meaning the program can run into size issues, or have to redeclare another array entirely.

Graphs were the fundamental class used to create the UAV program as it stores all location data and adjacent locations relative to each vertex. This class was the main class used for importing data and most importantly, traversal through locations. Both graph and graph vertex classes remain the same as it was in Practical 6, however some methods were removed, two new traversal methods were implemented and an update to its adjacency attributes were also added.

One of the new traversal methods that were added was Dijkstra's algorithm, an algorithm based off of Breadth-first search, where the method finds the shortest path from a start vertex to a destination vertex. Implementing this algorithm was much more effective in finding the shortest path to a destination vertex as some traversals sometimes prove that not always following the lowest weight is the fastest path to a destination. This method checks all possible methods of reaching the destination and returns the fastest path.

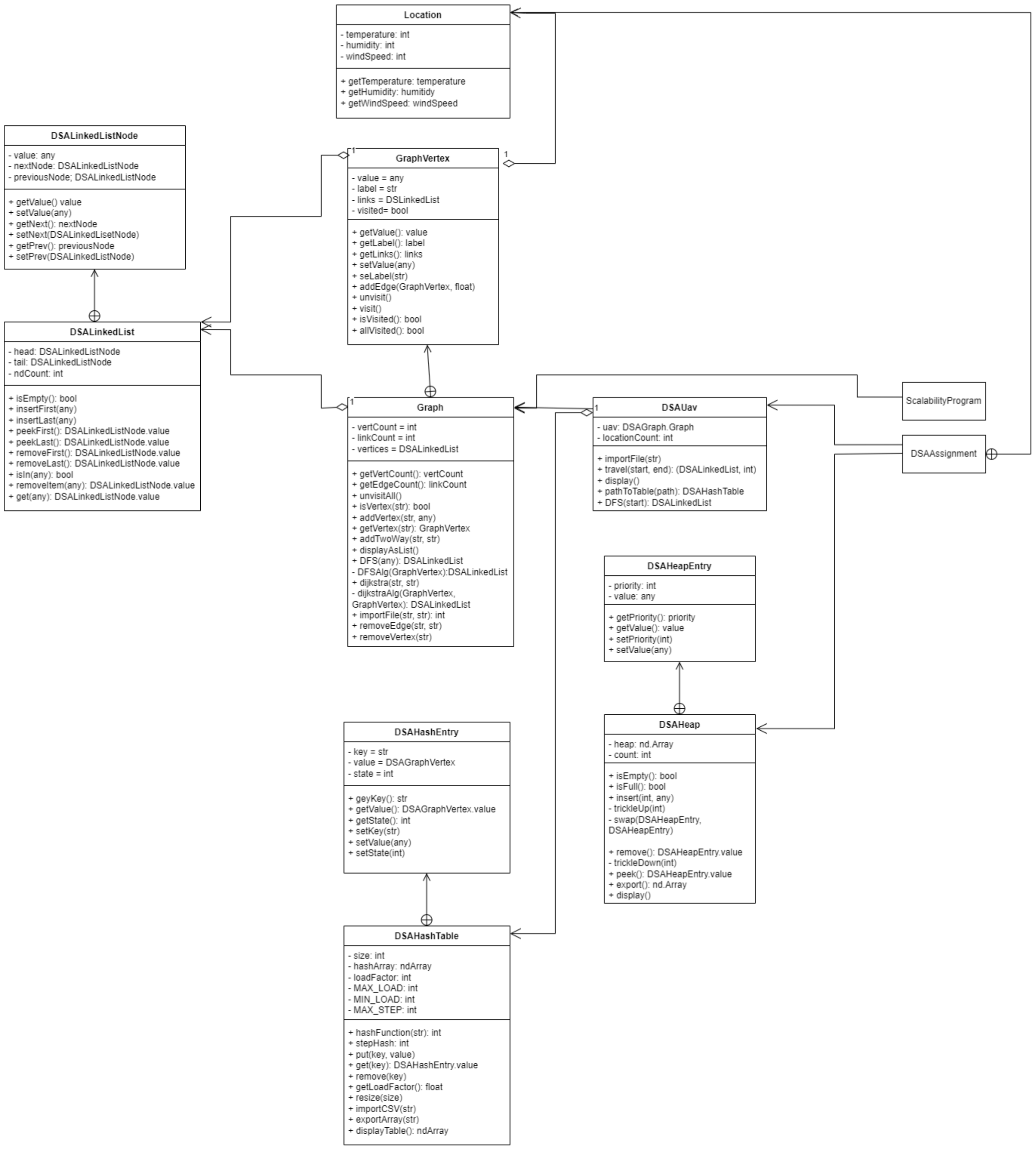
A* (A Star) is a new experimental algorithm used to find the path from a start vertex to a destination vertex. This algorithm is different to Dijkstra's algorithm as it is a heuristic algorithm, meaning that it estimates the path from the current vertex to the destination. This algorithm is also more optimal than Dijkstra's algorithm as it is under a condition called the "admissibility" of the heuristic function, meaning that this algorithm guarantees the shortest path. However, I was not able to fully implement the algorithm into the graph, resulting in an unfinished method along with the hamiltonianPath() method.

Hash tables were also an important class as it was the data structure used to store data based on the UAV's flight path to be accessed at O(1) speed. Linked lists would also be able to store data as well as hash tables, however, one major flaw of linked lists is that accessing items take O(N) time, where N is the amount of "nodes" there are in the list. Hash tables reduce that time to O(1) time as it uses a key, value pair, meaning that in order to access the data, a "key" is needed to access that value, rather than having to iterate through the list to find an item. The key is hashed into its own individual index, allowing the O(1) access speed. However, the only flaw with hash tables is that inability to store items with the same key, as it would cause errors by hashing to an already existing item in the table.

Heaps are used to sort an array of location based on its threat level. The threat level is a number ranging from 1 to 9, where 1 is the highest-level threat and 9 being the lowest. This threat level is used to create the itinerary for the UAV, where it has to fly over each high threat location in the shortest path possible. Though hash tables and linked lists are also a viable option to store threat levels, heaps are much more suited for this task as it automatically sorts the items in the array with its trickle up and trickle-down recursive methods.

The new DSAUav class is a wrapper class, wrapping up all traversal methods and data handling into one class. Instead of using Dijkstra's algorithm to return a path stored in a linked list, the wrapper returns two values instead, the total distance and the path. This saves time for both testing and production code, as the code needed to return the total distance is already inside the class itself. Some exception handling is also implemented in the class for the DSAAssignment.py program.

UML Class Diagram



Both linked list classes are required for both Graph and Graph Vertex classes to store vertex data and adjacent vertices for each vertex object. And for each Linked List, Graph, Hash Table, and Heap class, there is another class, Linked List Node, Graph Vertex, Hash Table Entry, and Heap Entry classes respectively.

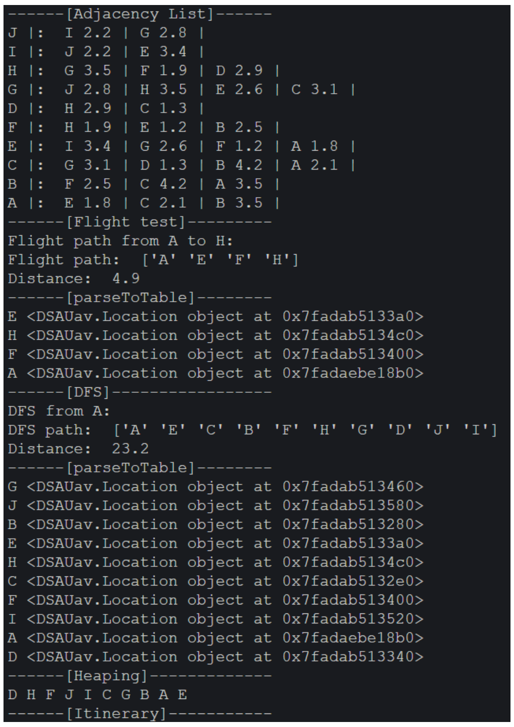
For the graph class, linked lists are used to behave as queues and stacks, as well as to store a list of Graph Vertex objects, and inside the Graph Vertex objects, linked lists are used to store adjacent vertices and their respective edges and weights. The Graph Vertex objects also contain a Location class, where it stores weather data such as temperature, humidity and wind speed.

The DSAUav class utilises both the Graph and Hash Table classes, as it performs both traversal and hashing in the same method. Most of the Graph classes methods are also utilised in the UAV class object.

Testing Methodology and Results:

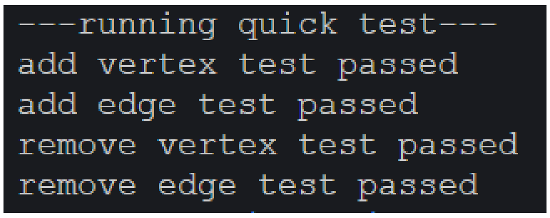
Both DSAAssignment.py and scalabilityProgram.py has a quick test function that quickly tests public methods, such as adding, removing vertices and edges, traversal methods, parsing, hashing, heaping. Apart from the quick test functions, the program was also tested on Jupyter Notebook, as shown in scripttest.ipynb, to test each method in real time and check for any inconsistencies inside the linked lists or arrays. Another file, testPage.py was used to test methods from classes outside of Jupyter Notebook to simulate production code. The UnitTestLinkedList.py file from Practical 4 has also been reused to test the functionality of the new Linked List class, as well as a test harness called testHarness.py, testing all methods from all classes in one file.

- An image of DSAAssignment.py 's quick test method, testing the UAV classes methods. The function tests methods such as outputs, traversal methods, hashing, and heaping.



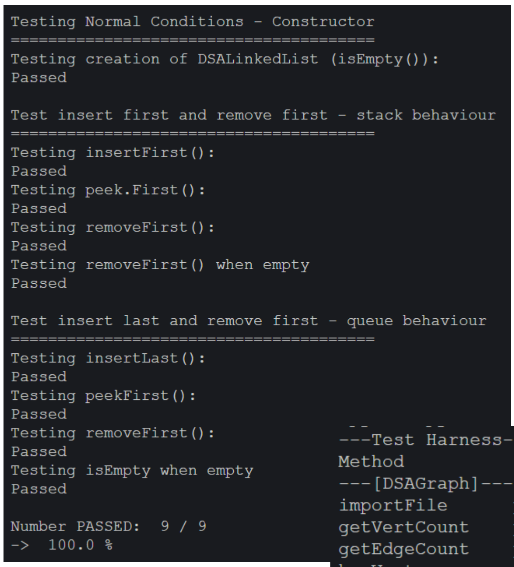
```
------[Adjacency List]------
J |:  I 2.2 | G 2.8 |
I |:  J 2.2 | E 3.4 |
H |:  G 3.5 | F 1.9 | D 2.9 |
G |:  J 2.8 | H 3.5 | E 2.6 | C 3.1 |
D |:  H 2.9 | C 1.3 |
F |:  H 1.9 | E 1.2 | B 2.5 |
E |:  I 3.4 | G 2.6 | F 1.2 | A 1.8 |
C |:  G 3.1 | D 1.3 | B 4.2 | A 2.1 |
B |:  F 2.5 | C 4.2 | A 3.5 |
A |:  E 1.8 | C 2.1 | B 3.5 |
------[Flight test]---------
Flight path from A to H:
Flight path:  ['A' 'E' 'F' 'H']
Distance:  4.9
------[parseToTable]--------
E <DSAUav.Location object at 0x7fadab5133a0>
H <DSAUav.Location object at 0x7fadab5134c0>
F <DSAUav.Location object at 0x7fadab513400>
A <DSAUav.Location object at 0x7fadaebe18b0>
------[DFS]-----------------
DFS from A:
DFS path:  ['A' 'E' 'C' 'B' 'F' 'H' 'G' 'D' 'J' 'I']
Distance:  23.2
------[parseToTable]--------
G <DSAUav.Location object at 0x7fadab513460>
J <DSAUav.Location object at 0x7fadab513580>
B <DSAUav.Location object at 0x7fadab513280>
E <DSAUav.Location object at 0x7fadab5133a0>
H <DSAUav.Location object at 0x7fadab5134c0>
C <DSAUav.Location object at 0x7fadab5132e0>
F <DSAUav.Location object at 0x7fadab513400>
I <DSAUav.Location object at 0x7fadab513520>
A <DSAUav.Location object at 0x7fadaebe18b0>
D <DSAUav.Location object at 0x7fadab513340>
------[Heaping]-------------
D H F J I C G B A E
------[Itinerary]-----------
```

- An image of scalabilityProgram.py 's quick test method.

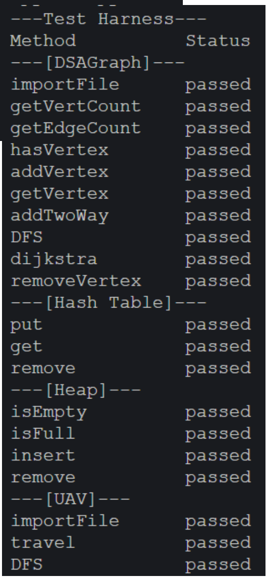


```
---running quick test---
add vertex test passed
add edge test passed
remove vertex test passed
remove edge test passed
```

- UnitTestLinkedList.py



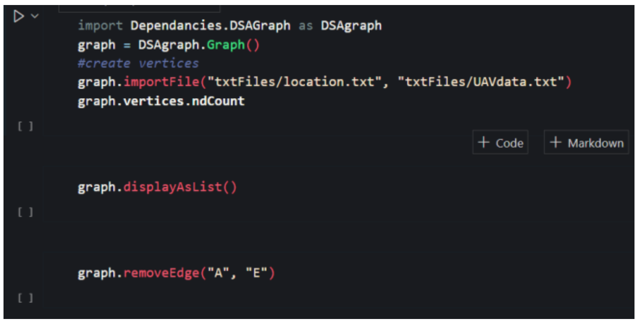
```
Testing Normal Conditions - Constructor
=======================================
Testing creation of DSALinkedList (isEmpty()):
Passed

Test insert first and remove first - stack behaviour
=======================================
Testing insertFirst():
Passed
Testing peek.First():
Passed
Testing removeFirst():
Passed
Testing removeFirst() when empty
Passed

Test insert last and remove first - queue behaviour
=======================================
Testing insertLast():
Passed
Testing peekFirst():
Passed
Testing removeFirst():
Passed
Testing isEmpty when empty
Passed

Number PASSED:  9 / 9
->  100.0 %
```



```
---Test Harness---
Method          Status
---[DSAGraph]---
importFile      passed
getVertCount    passed
getEdgeCount    passed
hasVertex       passed
addVertex       passed
getVertex       passed
addTwoWay       passed
DFS             passed
dijkstra        passed
removeVertex    passed
---[Hash Table]---
put             passed
get             passed
remove          passed
---[Heap]---
isEmpty         passed
isFull          passed
insert          passed
remove          passed
---[UAV]---
importFile      passed
travel          passed
DFS             passed
```

An image of scripttest.ipynb.

The notebook is used to test methods throughout testing in real time, rather than having to test a whole file every time.



```
import Dependancies.DSAGraph as DSAgraph
graph = DSAgraph.Graph()
#create vertices
graph.importFile("txtFiles/location.txt", "txtFiles/UAVdata.txt")
graph.vertices.ndCount

graph.displayAsList()

graph.removeEdge("A", "E")
```

Reflection:

Reflection: Areas for Improvement

Upon reflecting on my recent work, there several aspects that could have been improved. Firstly, due to time constraints, I was not able to implement the A* algorithm. I realised that utilising a linked list to store adjacent vertices may not have been the most efficient choice. Opting for a hash table instead could have resulted in quicker access and enhanced overall performance, as hash tables have O(1) time and better removal methods, this could have made the implementation of the scalability program much faster, allowing me to work on better algorithms.

References

- Practical 4 – Linked Lists

- Practical 6 – Graphs

- Practical 7 – Hash Tables

- Practical 8 – Heaps

- Dijkstra’s Algorithm:
  - https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

- A* Algorithm:
  - https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm