Data Structures and Algorithms

Final Assignment

Semester 1, 2023

Malith Pramuditha 20926076

Task 1: Creating a Graph

In order to create a graph representation of the region of interest, the computer reads an input file named "location.txt" in this job. The graph's number of vertices, edges, beginning and terminating vertices, and weight of each edge are all specified in the file. As output, the software creates an adjacency list representation of the graph.

Task 2: Implementing BFS and DFS

In order to complete this task, the graph must be traversed using the depth-first search (DFS) and breadth-first search (BFS) methods. The "UAVdata.txt" file, which contains information on each location's temperature, humidity, and wind speed, is used as input by the software. Using BFS, the computer can determine the shortest route between any two points and gather relevant information for each place along the way.

Task 3: Insert, Delete, and Searching Operations

This work entails developing actions to insert, remove, and search places inside the graph in order to improve the program's scalability. For carrying out these actions whenever you choose, the application offers an interactive menu.

Task 4: Hashing the Data

A hash table is used to effectively store and retrieve the data related to each place. Based on the geographical identification, the application uses a hash table data structure to swiftly seek up the data. The effectiveness of utilizing a hash table in comparison to looking through a list or array is contrasted and explored.

Task 5: Using a Heap

A heap data structure is used to track the locations with the greatest danger of bushfires. The heap makes it easy to obtain the greatest risk value from a group of risks. The software guarantees that the heap is updated with the most risky places by adding new risk values to it.

Task 6: Providing an Itinerary

This activity entails adjusting UAV flight routes depending on distances and threats connected to various sites. The algorithm creates an itinerary that prioritizes regions with a high risk of bushfires while using the least amount of energy possible. Other algorithms besides BFS and DFS may be used by the application to carry out this optimization.

Heap

A binary tree-based data structure that meets the heap property is called a heap. According to the heap property, each parent node for a max heap must have a value larger than or equal to its children, whereas each parent node for a min heap must have a value lower than or equal to its children. Because the root node of a heap always contains the maximum or minimum value, heaps are frequently used to effectively extract the maximum or minimum element from a group of data.

Graph

A graph is a non-linear data structure made up of a collection of vertices (also known as nodes) linked together by edges. Graphs are used to show the connections between different items or things. They may be divided into directed graphs (which have edges that point in a certain direction) and undirected graphs (which don't have edges that point in a particular direction). Graphs are often used in many different applications, including those that show networks, social relationships, road maps, and other things. They may be visited to examine their attributes or discover routes between vertices using algorithms like Breadth-First Search (BFS) or Depth-First Search (DFS).

Hash Table

A hash table, commonly referred to as a hash map, is a type of data structure that makes it easy to insert, remove, and retrieve key-value pairs. It maps keys to an array index, where the relevant value is kept, using a hash function. Hash tables are extremely effective for handling enormous datasets because they offer constant-time average case complexity for these operations. The common methods for resolving collisions, which happen when several keys hash to the same index, are chaining (using linked lists) and open addressing (searching for alternate places). Databases, caches, and symbol tables are a few examples of applications that frequently employ hash tables for quick lookup.

Linked List

A linked list is a type of linear data structure made up of a series of nodes, each of which has a reference (or link) to the node after it in the list. Unlike arrays, linked lists can increase or contract in size dynamically and do not require contiguous memory allocation. A linked list's head node is the first node, and its end node normally contains a reference to null. Linked lists provide effective insertion and deletion operations at any location, but sequential list traversal is required to reach entries in the center of the list. There are several kinds of linked lists, such as singly linked lists (where each node has a connection to the node after it) and doubly linked lists (when each node has links to both the next and previous nodes).

Queue

First-In-First-Out (FIFO) is a linear data structure that governs queues. The rear or enqueue action, which adds items to one end of the collection, and the front or dequeue operation, which removes members from the other end, represent the collection of elements. Existing parts are removed from the front and new ones are added to the back. In situations like job scheduling, breadth-first search, and simulations, where the order of processing or accessing items is crucial, queues are frequently employed.

Stack

Linear data structures that adhere to the Last-In-First-Out (LIFO) concept include stacks. When pieces are added or withdrawn, they only affect the top end of the collection, which it represents. Push operations are used to introduce items, and pop operations are used to remove them. In addition, the top element may be viewed without removing it by using a peek technique. Applications like expression evaluation, function call management, depth-first search, and undo-redo capabilities all make extensive use of stacks.

Dijkstra algorithm

In a weighted directed or undirected network, the single-source shortest route issue is resolved by Dijkstra's method.

The method operates by keeping track of a priority queue of vertices, each of which is assigned a speculative distance value. All other vertices' distance values are initially set to infinity, while the source vertex's distance value is first set to zero. The technique gradually adjusts (relaxes) the distances between nearby vertices while choosing the vertex with the lowest distance value from the priority queue.

Until the goal vertex is reached or all vertices have been visited, the procedure keeps going.The method selects the vertex with the least tentative distance at each iteration and investigates its neighboring vertices. A vertex's distance value is updated if a shorter path to it is discovered.

If there are no negative-weight edges in the graph, Dijkstra's method will always discover the shortest route between the source vertex and every other vertex. The shortest paths from the source vertex to all other vertices are kept in a data structure dubbed a "shortest path tree" by the method.

DSAGraph class and DSAGraphNode class

The adjacency list implementation used by the DSAGraph class to describe a graph data structure. It provides for the addition of vertices and edges, the verification of vertex existence, the retrieval of vertex and edge counts, the verification of vertex adjacency, the retrieval of neighboring vertices, and the presentation of the graph as a list.

A node in the graph is represented by the DSAGraphNode class. Each node in the network includes a label, a list of nodes that it neighbors, a reference to the node after it, and a count of those nodes. It offers ways to add neighbors, check whether there are neighbors, get neighbors, and show the neighbors.

Together, these classes offer a fundamental framework for designing and working with graphs. While the DSAGraphNode class represents specific nodes within the graph, the DSAGraph class acts as the primary interface for interacting with the graph. To hold the vertices and their relationships, the DSAGraph class keeps a linked list of DSAGraphNode objects.

DSAHashTable and DSAHashEntry class

A hash table entry is represented by the DSAHashEntry class. It includes a key-value pair as well as a status that indicates whether the entry is free, previously used, or presently utilized. It offers ways to get at and change the entry's key, value, and state.

A hash table data structure is represented by the DSAHashTable class. The hash entries are stored using an implementation based on an array. The class has methods for adding key-value pairs to the hash table, accessing values linked to keys, determining if a key already exists in the table, deleting key-value pairings, and expanding the hash table.

For enlarging the hash table, the DSAHashTable class additionally offers assistance methods like is_prime and next_prime that may be used to identify prime values. It computes the hash value for a given key using a hashing algorithm and manages collisions using linear probing with a step size of 1.

When the number of entries reaches a certain percentage of the size of the table, the class maintains a load factor threshold to dynamically enlarge the hash table. Additionally, the table contains a shrink factor threshold that adjusts the size of the table when the number of entries is below a predetermined level.

DSAHeap class and DSAHeapEntry class

A heap data structure entry is represented by the DSAHeapEntry class. It has a matching value and a priority value. The class offers ways to get at and change the entry's priority and value.

A binary min-heap data structure is implemented by the DSAHeap class. It stores the heap items using an array-based representation. The class has methods for adding items to the heap, removing the item with the lowest priority from the heap, and determining if the heap is empty.

When a parent node's priority is lower than or equal to that of its child nodes, the heap preserves the heap property. When restoring the heap property, the _bubble_up function swaps an element with its parent if necessary after insertion. If necessary, the _bubble_down function swaps an element with its smallest child after removal in order to restore the heap property.

DSALinkedlist and DSALinkedListIterator class

A double linked list data structure is implemented by the DSALinkedList class. It consists of discrete nodes, represented by the DSAListNode class, each of which has a value as well as pointers to the nodes immediately before and after it.

The first and last nodes of the linked list may be removed, items can be added at the beginning or end, the values of the first or last node can be retrieved, and the DSALinkedList class has methods to manage the linked list. It also has techniques for determining if the list is empty or filled.

A head pointer and a tail pointer are used by the class to maintain track of the first and last nodes, respectively. The relevant references are updated when a new node is added in order to preserve the linked list's proper hierarchy and linkages.

An iterator implementation for the DSALinkedList class is the DSALinkedListIterator class. It permits using a for loop or other iterable operations to repeatedly iterate through the linked list's elements. Up until it reaches the end of the list, the iterator goes from the head of the list to the subsequent node.

DSAStack

The DSAStack class uses a linked list to implement a stack data structure. The stack's components are stored using the DSALinkedList class.

The DSAStack class offers ways to work with the stack, including peek to get the top element without removing it, pop to remove and return the top element, and push to add an element to the top of the stack. To determine whether the stack is empty, the class additionally has an isEmpty function.

The associated methods of the underlying linked list are used to implement the stack operations. Push, for instance, uses the insert_first function to add an element to the beginning of the linked list, whereas pop uses the remove_first method to remove and return the first member.

A for loop or other iterable operations may be used to iterate over the elements in the stack using the DSAStack class's __iter__ function, which is also defined. It gives back an iterator object that loops through the linked list's items.

DSAqueue

A queue data structure employing a linked list is represented by the DSAQueue class. The queue's items are stored using the DSALinkedList class.

The DSAQueue class offers queue manipulation options. By utilizing the insert_last method to place an element at the end of the linked list, the enqueue function adds that element to the queue. By utilizing the remove_first method to remove the first element from the linked list, the dequeue function removes and returns the element at the front of the queue. Using the peek_first function of the underlying linked list, the peek method enables accessing the element at the front of the queue without destroying it. The linked list's is_empty method is used by the is_empty method to determine whether the queue is empty.

The __iter__ function, which produces an iterator object that enables iterating through the queue's items in the sequence they would be dequeued, is another feature of the DSAQueue class. The iterator of the underlying linked list is used to do this.

task1.py

It imports the required modules' versions of the classes DSAGraph, DSAGraphNode, DSALinkedList, DSALinkedListIterator, and DSAQueue.

By iterating over the graph's nodes and their neighbors, the display_graph function takes a graph as an input and outputs a representation of the graph's adjacency list.

The graph is expanded by the insert_edge function by adding an undirected edge between two nodes. The nodes are added to the graph after being first checked to see whether they already exist. The adjacency sets of both nodes are then updated, and the edge is added.

A graph edge connecting two nodes is eliminated using the delete_edge function. If the edge exists between the nodes, it is verified and taken off of both nodes' adjacency sets.

The search_edge function determines if a graph edge connects two nodes and reports the outcome.

The user is prompted by the code to provide the filename from which to read the graph data.

It reads each line from the opened file according to the provided path, then executes the insert_edge method to add the edges to the graph dictionary.

To display the original graph, it invokes the display_graph function.

It then moves into a looping interactive menu where the user can select from many options:

1: Add a border. runs the insert_edge method while asking the user to enter the node labels.

2: Eliminate a corner. runs the delete_edge method while asking the user to provide the node labels.

3: Look for a cutting edge. runs the search_edge method while asking the user to enter the node labels.

4. use the display_graph method to report the graph's current state.

5: Quit the application. ends the application and exits the menu loop.

It handles the FileNotFoundError exception and generates an error message if the requested file cannot be located.

graph.py

A graph and UAV (Unmanned Aerial Vehicle) data are read from files by the programme. The next step is to use a Breadth-First Search (BFS) to locate the graph's shortest path between any two points risk factor or weight age not included in BFS in when find shortest path. For each node along the shortest path, the UAV data is shown. Finally, it displays the UAV data for the full graph using Depth-First Search (DFS).

In order to determine the shortest route between a start node and an end node in the graph, the bfs function uses breadth-first search. By putting neighbors of visited nodes in a queue, it

traverses the graph breadth first. When it reaches the end node, it halts its search and returns the shortest path.

The depth-first search method is used by the dfs function to examine the whole graph. Each node's neighbors are visited repeatedly, marked as visited, and the UAV data related to each node is printed.

Hashtable.py

The hash table data structure is implemented in the code utilizing distinct chaining to avoid collisions. It offers the ability to add key-value pairs to the hash table, retrieve values using keys, and delete key-value pairs from the hash table.

Graph data and UAV (Unmanned Aerial Vehicle) data may be read from files and stored in separate hash tables using functions in the code. While the UAV data provides details about the coordinates of each site, the graph data depicts linkages between locations.

The user is prompted to provide the file names for the data and graph in the code's main section. The data is subsequently read into hash tables from the files. In order to show the x, y, and z coordinates associated with the place, it then asks the user to provide a location and pulls the pertinent data from the hash table.

Overall, the code shows how to effectively store and retrieve data based on unique keys using a hash table.

heap.py

An array is used in the code to implement a heap data structure. In order to maintain the heap property, the DSAHeap class has methods to push objects onto the heap, pop the smallest item off the heap, verify if the heap is empty, and carry out internal activities like bubbling up and bubbling down.

User provide the file names. Based on temperature, humidity, and wind speed, the algorithm extracts information from a file (data.txt) and determines the danger level for each place. The risk level is calculated by giving each element a score and adding them together. Then, using the negative risk level as the priority, the locations and their danger levels are piled together.

By removing things from the heap and showing their priority and area, the algorithm produces the areas with the highest risk value. For the top 10 things in the pile, it repeats this procedure.

The user is then prompted by the code to provide new values for the temperature, humidity, and wind speed. The new data is pushed onto the heap after the risk level has been calculated for it. By selecting objects from the heap and highlighting their priority and area, it then generates the updated greatest risk value areas.

iter.py

DSAGraph classThis class uses an adjacency list to describe a weighted graph. There is a function called add_edge that allows you to add edges between nodes with the appropriate weights.

The read_graph method produces an instance of the DSAGraph class by reading the graph data from a file. The first line of the file is used to extract the number of nodes and edges, after which the add_edge function is used to add the edges to the graph.

Dijkstra's approach is implemented by the dijkstra function, which seeks out the shortest routes between a specified start node and every other node in the graph. It updates the distances as it moves along the graph, maintaining a priority queue (heap) of nodes depending on their present distance from the start node.

Reading UAV dataThe code requests the file names for the graph data and UAV data from the user. The UAV data is read from the designated file, the danger level for each region is determined using the temperature, humidity, and wind speed, and the areas and risk levels are then stored in a heap (priority queue).

The code prints the locations with the greatest risk value by removing the elements from the heap and showing their names.

User input required: The user is prompted by the code to enter the quantity of UAVs and their beginning locations.

The dijkstra function is called by the code to produce the flight routes for each UAV. The generated pathways are kept in a list.

Printing flight routes: Beginning from each UAV's corresponding beginning point, the code prints the flight paths for each UAV. It outputs the path in the format "Node - Node -... - Node," removing the beginning point from the path list and appending it at the end for a full cycle.

testing methodology and results

task1.py

The offered code enables interactive manipulation of an adjacency list-based graph data structure. It displays the basic graph after reading graph data from a file and adding edges to the graph.

Options for adding edges, removing edges, searching for edges, displaying the graph, and quitting the software are all available on the interactive menu. By keying in the relevant number, the user can select an operation.

You can prepare a text file with graph data to test the code. Two node labels denoting an edge between each pair of nodes should be contained on each line of the file. When asked, run the software and input the filename.Upon running the program and entering "graph.txt" as the filename,

Following that, you have the option to create a new edge between two nodes, remove an existing edge, look for an edge, or show the graph by selecting one of the choices from the menu. The chosen operation will be carried out by the software, and the graph will be updated appropriately.

In order to verify the functioning of edge insertion, deletion, and search, the software is run, input files containing varied graph data are provided, and various operations are carried out. The displayed graph and the program's output messages, which confirm successful operations or alert users to issues, both show the outcomes.

Keep in mind to adhere to the graph data file input format and to interact with the graph using the menu choices in accordance with your testing needs.

```
Input filename: location.txt
Edge (10, 15) inserted.
Edge (A, B) inserted.
Edge (A, C) inserted.
Edge (A, E) inserted.
Edge (B, C) inserted.
Edge (B, F) inserted.
Edge (C, D) inserted.
Edge (C, G) inserted.
Edge (D, H) inserted.
Edge (E, F) inserted.
Edge (E, G) inserted.
Edge (E, I) inserted.
Edge (F, H) inserted.
Edge (G, H) inserted.
Edge (G, J) inserted.
Edge (I, J) inserted.
Adjacency List:
10: 15
15: 10
A: B, E, C
B: C, F, A
C: B, D, A, G
E: I, F, A, G
F: B, H, E
D: H, C
G: J, H, E, C
H: D, F, G
I: J, E
J: I, G

Menu:
1. Insert
2. Delete
3. Search
4. Display Graph
5. Exit
```

graph.py

A graph and UAV data are read from input files by the given code. The user may then utilize breadth-first search (BFS) to determine the shortest route between any two points on the graph, and depth-first search (DFS) to show the UAV data for the whole graph.

User provide theTwo input files, "graph.txt" holding the graph data and "data.txt" having the UAV data, are required to test the code. For proper execution, these files' formats are crucial.

The code is executed after reading the data and graph from the input files. In order to identify the quickest journey, the user is then required to specify the beginning and finishing locations.

The code first verifies that the locations input are correct before running BFS to determine the shortest route between them. It shows the path and the UAV data (temperature, humidity, and

wind speed) for each node in the path if one is discovered. It shows a matching message if no path is detected.

The algorithm then runs DFS to traverse the whole graph before displaying the UAV data for each node.

Make sure that the input files have correct data and formatting before running the code. To find several shortest pathways, you can select various beginning and finishing points. Check to make sure the shortest path and UAV data are shown properly. Check to see if the DFS function presents the UAV data for the complete graph in a right manner as well.

To get reliable results, make sure that the input files and user inputs follow the anticipated formats and specifications.

```
Enter your choice (0-5): 2
Enter the graph file name: location.txt
Enter the data file name: UAVdata.txt
Input the starting location: A
Input the ending location: J
Shortest path of A and J: ['A', 'C', 'G', 'J']
A: temperature=32C, humidity=45, wind speed=90km/h
C: temperature=38C, humidity=55, wind speed=75km/h
G: temperature=42C, humidity=60, wind speed=50km/h
J: temperature=33C, humidity=35, wind speed=60km/h
UAV data for the graph:
A: temperature=32C, humidity=45, wind speed=90km/h
B: temperature=26C, humidity=50, wind speed=35km/h
C: temperature=38C, humidity=55, wind speed=75km/h
D: temperature=45C, humidity=30, wind speed=80km/h
H: temperature=36C, humidity=25, wind speed=95km/h
F: temperature=31C, humidity=20, wind speed=85km/h
E: temperature=29C, humidity=40, wind speed=65km/h
G: temperature=42C, humidity=60, wind speed=50km/h
J: temperature=33C, humidity=35, wind speed=60km/h
I: temperature=27C, humidity=50, wind speed=40km/h
Main Menu:
```

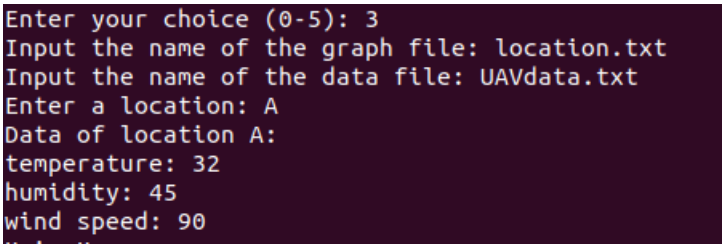
Hashtable.py

This program provides a hash table data structure and makes use of it to read input files containing graph and UAV data. It enables the user to get the UAV data for a certain area.

The file names for the data files and the graph are retrieved from the user in the code's main section. The data for a specific place given by the user is then obtained and shown after the graph and data have been read from the files using the appropriate procedures.

You need input files with legitimate data in the required formats in order to test the code. Make sure the data file has location data with the format "location x y z" and the graph file contains edge data with the format "start end weight".

When prompted, run the code and supply the file names. Next, provide the address of the site for which you wish to get UAV data. If the data is discovered, the code will show the x, y, and z values for that place. If not, it will state that there is no data for the location.For accurate results, be careful to enter the relevant file names and location information.



heap.py

user need to input data.txt file with area data in the type "area temperature humidity wind_speed" for each line if you want to test the code.

The testing approach you may use is as follows:

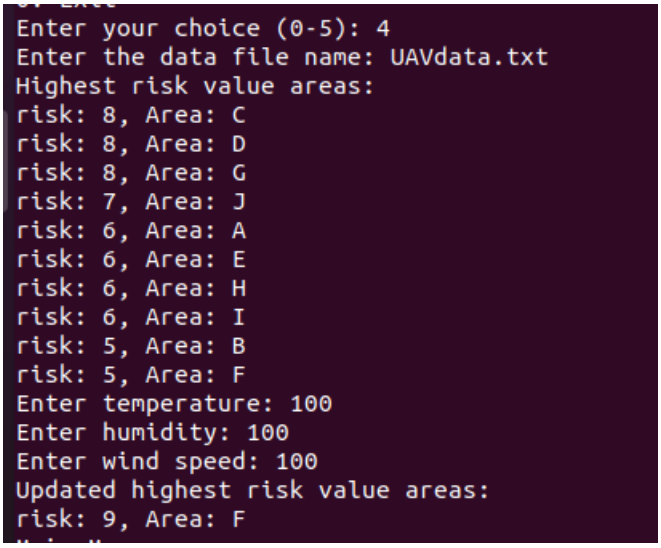
Create a data.txt file with area data for several locations, with each line representing data in the format "area temperature humidity wind_speed" for a single location. Make sure that the temperature, humidity, and wind speed are represented numerically on each line.

Run the program and look at the results. Based on the supplied data, it will provide the places with the highest risk value.

Enter new information for the temperature, humidity, and wind speed when requested. Make that the values are in numerical format and fall inside an acceptable range.

When new data is entered, the code will update the highest risk value regions and recalculate the risk level for the new data before pushing it into the heap.

Verify that the code runs flawlessly and generates the desired results. Verify the code's accuracy by going over its logic and the computations that were made.



iter.py

putting the graph data file in Make a text file with the graph data in it. The number of nodes and edges in the graph should be stated in the first line. The syntax "start end weight" should be used to express each succeeding line as an edge. Give the file a suitable name before saving it.

the UAV data file as input: Make a text file with the data from the UAV. Spaces should be used to separate the relevant temperature, humidity, and wind speed numbers for each location represented by a line. Give the file a suitable name before saving it.

Run the program: When prompted, run the code and supply the file names. The program will read the graph and the UAV data, compute the risk factors for each location, and provide fly routes for the number of UAVs that have been specified.UAV will flight to the highest risk areas.

Examine the result: Based on the determined danger levels, the code will output the places with the greatest risk value. You will next be prompted to input the quantity and starting positions of UAVs. It will show the flight routes for each UAV after optimization.

test using a little graph and data from a UAV: Make a tiny data file representing a graph with a few nodes and edges. A UAV data file with a few regions and their related values should be prepared. Run the code to see if it creates legitimate flight routes and the anticipated greatest risk value zones.

Create a bigger graph data file with more nodes and edges to create a realistic scenario when testing with a larger graph and UAV data. A UAV data file should be prepared with enough regions. Run the code to see if it provides logical flight pathways and efficiently completes the optimization.

Review the output to make sure it corresponds with the anticipated outcomes after executing the code and carrying out the tests. Check to see whether the locations with the highest risk values have been appropriately detected and if the resulting flight paths make sense in light of the graph and the UAVs' beginning positions.

```
Input the number of UAVs: 3
Input the starting position for UAV 1: A
Input the starting position for UAV 2: B
Input the starting position for UAV 3: C

UAV_1:
D <- F <- I <- B <- J <- E <- H <- G <- C <- A <-
UAV_2:
D <- A <- F <- I <- J <- E <- H <- G <- C <- B <-
UAV_3:
D <- A <- F <- I <- B <- J <- E <- H <- G <- C <-
```

how this implementation might be improved

task1.py

Use a set to implement the adjacency list: A list is currently being used to implement the adjacency list. It would be better to use a set to record the neighbors of each node because duplicate edges are not permitted and the order of the neighbors is irrelevant. As a result, testing for the existence of an edge will be more effective, and adding and deleting neighbors will have constant-time complexity.

Add suitable error handling and input validation to deal with situations like improper user input in the interactive menu and invalid filenames and node labels. Verify user input to make sure node labels are original and devoid of any problematic characters.

Make variable and function names meaningful: By giving variables and functions names that are more descriptive, its may make this code easier to read. The code will be simpler to comprehend and maintain as a result.

Think about utilizing a data structure library: Instead of writing linked lists and queues from scratch, you may use the Python language's built-in data structure classes or libraries, such as collections.deque for queue implementation. This will prevent this code from having to start from scratch and guarantee effective implementations.

Add more graph operations: Depending on the needs of your application, you might think about adding more graph operations, like calculating metrics like a node's degree or the number of

connected components, traversing the graph (DFS, BFS), and finding the shortest path between nodes.

These recommendations are meant to enhance user experience, performance, and code structure. Cleaner and more reliable code will follow from putting these changes into practice.

What further investigations and/or extensions could be added?

Consider utilizing libraries or other tools for graph visualization to see the graph. By giving the graph's structure a visual representation, this can facilitate comprehension and analysis.

Implement techniques for serializing and deserializing a graph to and from a file or a string representation. This is beneficial for both storing and loading graph data as well as moving the graph between platforms.

Performance optimization: Examine how well the graph operations work and look for any potential bottlenecks. In order to increase the effectiveness of frequent graph operations, think about streamlining the code by leveraging the proper data structures or algorithms.

graph.py

Add adequate error handling and input validation to deal with situations like incorrect filenames, missing data in input files, and incorrect user input. Verify user input to make sure that the starting and terminating points are legitimate graph nodes.

Effectively use data structures: Instead of representing the network with a dictionary of dictionaries, think about using a more effective data structure, such an adjacency list or an adjacency matrix. This will enhance space efficiency and make graph operations simpler.

Implement graph algorithms as distinct methods: Within the Graph class, construct separate methods for the BFS and DFS algorithms rather than incorporating them directly into the main code. As a result, modularity will be enhanced, making it simpler to test and reuse these methods.

Add more graph operations: Depending on the demands of your application, you might think about adding more graph operations to the Graph class, including calculating the shortest path using different algorithms like Dijkstra's algorithm, finding the minimum spanning tree, and topological sorting.

Give clear error messages: Enhance the error messages to give more detailed information about the errors detected. Users will find it easier to comprehend and fix the problems as a result.

What further investigations and/or extensions could be added?

Support for Weighted Graphs: Add weighted graph support to the graph implementation. The current algorithm assumes an unweighted graph with equal weights for each edge. Weighted edges can be handled by modifying the graph data structure and algorithms, enabling more accurate and flexible graph representations.

Implement a feature that allows you to graphically depict the graph and its attributes, such as graph visualization. It is simpler to evaluate and comprehend the structure of the network by using visual representations of the graph created using graph visualization frameworks.

Hashtable.py

Enhance the insert method: The key is currently checked for existence in the hash table using a linear search. To enhance the insertion performance, think about employing a data structure like a linked list or a binary search tree in each bucket. By doing this, the time complexity can be lowered from O(n) to O(log n) or, in the case of a linked list, O(1).

Add appropriate error handling and input validation to deal with situations like invalid filenames and inappropriate file formats. Verify user input to make sure it follows the desired format.

Improve error messages to include more detailed information about the detected errors in order to make them more relevant. Users will find it easier to comprehend and fix the problems as a result.

Add more hash table operations: Depending on the demands of your application, you may want to add more hash table operations, such as the ability to remove, update, or repeatedly iterate over all hash table components. The hash table's usability and functionality may be improved by these procedures.

What further investigations and/or extensions could be added?

Put a hash table iterator in place: Make a class of iterator that enables iterating through each key-value pair in the hash table. This can be helpful for processes like iterating, filtering, or changing the hash table's component parts.

Managing key-value collisions when there are several possible values In the existing system, every key is thought to have a distinct value. If a key can have more than one value, you can change the implementation to prevent collisions by storing more than one value in the hash table. This can entail storing several values connected to the same key in an array or linked list of data structures.

heap.py

Add suitable error handling and input validation: Handle situations like invalid file names and wrong file formats by performing error handling and input validation as necessary. Verify user input for consistency with the desired format and deal with any unexpected exceptions that can arise during file operations.

Implement a priority queue: Rather of using a straightforward heap, you might want to use a priority queue data structure if you need to frequently update the risk levels or dynamically prioritize regions based on their danger levels. With each element having a priority assigned to it, a priority queue enables efficient insertion and extraction of elements.

What further investigations and/or extensions could be added?

Add methods to serialize the heap to a file and deserialize it to restore it to its initial state in order to implement serialization and deserialization. You may persist the data between program executions by using this to save and load heap data.

Examine the efficiency and complexity of heap operations by performing a performance analysis to find out how time-consuming they are and locate any possible bottlenecks. To comprehend the scalability properties of the heap implementation, think about analyzing its effectiveness for a range of input sizes.

Iter.py

Add a calculator function to aid with risk level calculation: To increase code modularity and reuse, isolate the logic for risk level calculation into a different function. The danger level will be returned based on the inputs of temperature, humidity, and wind speed.

Print flight trajectories using a helper function that you implement: Make a separate function that outputs the fly routes for each UAV after receiving the beginning positions and flight pathways as inputs. By doing so, code duplication in the primary logic will be removed and code readability will be improved.

Add suitable error handling and input validation: Handle situations like invalid file names and wrong file formats by performing error handling and input validation as necessary. Verify that user input follows the intended format by validating it, and deal with any potential exceptions that may arise during file operations or input processing.

What further investigations and/or extensions could be added?

Enhance graph operations: Examine the effectiveness of graph operations and think about making them even better. You may look at methods like graph compression, parallel computing, or specialized data structures to increase the effectiveness of graph-related tasks depending on the size and complexity of the graph.

Create an interactive user interface that enables users to enter data, view the graph, alter the settings, and view the results. This would improve the code's usability and accessibility and make it simpler for users to engage with the system and get insights.

Consider scalability: To boost scalability and manage the higher computational demands, look at distributed computing frameworks or cloud-based solutions if the code must handle larger datasets or more complex scenarios.

compare and discuss the efficiency of using the hashtable than searching through a list or array.

In some situations, using a hashtable, sometimes referred to as a dictionary or hashmap, can be significantly more efficient than searching through a list or array. When contrasting the effectiveness of utilizing a hashtable to browsing through a list or array, keep the following considerations in mind:

Constant-time lookup: The average-case constant-time complexity of hashtable lookup is O(1). This indicates that the time it takes to obtain an element is essentially consistent, regardless of the size of the hashtable. On the other hand, searching through a list or array requires iterating over each element until the required element is located or the end of the list is reached, which results in a linear-time complexity of O(n).

Key-value association efficiency: Hashtables are built for key-value association efficiency. A hashtable stores each element according to its distinct key, and the hashing algorithm makes

sure that the key maps to a certain index in the underlying array. This makes it possible to quickly retrieve the value linked to a certain key without having to go through the full collection. Comparing each element in turn until a match is discovered is required while searching through a list or array.

RAM use trade-off: To store the hash table structure and handle collisions, hashtables often need more RAM. The effectiveness and memory use of the hashtable might be affected by its size. The amount of memory overhead may be smaller while looking through a list or array since no additional data structures are needed. Therefore, utilizing a list or array could be more effective in situations when memory utilization is a problem.

Managing enormous datasets: As the amount of the dataset grows, it becomes more obvious how much more efficient hashtables are than searching through a list or array. The linear search makes searching through a big list or array more time-consuming but keeps the hashtable lookup time constant.

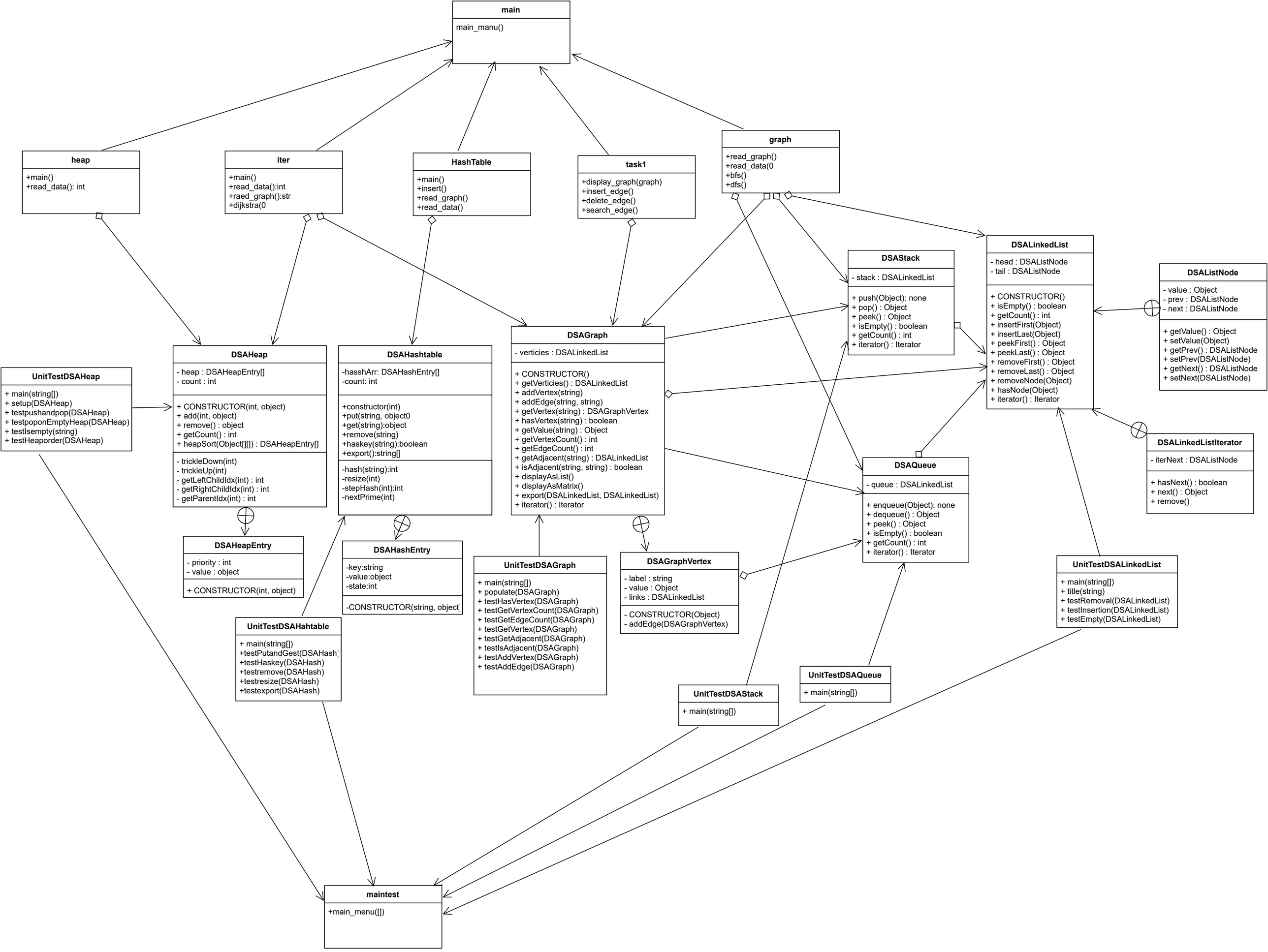
Sample Traceability Matrix

Feature / Function	Requirements	Design	Test
Graph functions			
Add vertex	add vertex to the Graph	addVertex() in Graph.py class and tested in testaddvertex() in UnittestDSAGraph.py	[PASSED] added vertex
Add edge	add edge to the Graph	addedge() in Graph.py class and tested in testaddEdge() in UnittestDSAGraph.py	[PASSED] added Edge
Has vertex	Check vertex in Graph	HasVertex() in Graph.py claas and tested in testaddvertex() in UnittestDSAGraph.py	[PASSED] checked vertex

Get vertex count	Get the vertex count in Graph	getVertexcount() in Graph.py claas and tested in testaddvertex() in UnittestDSAGraph.py	[PASSED]
Get edge count	Get the edge count in Graph	getEdgecount() in Graph.py claas and tested in testgetEdgecount() in UnittestDSAGraph.py	[PASSED]
Is adjacent		Isadjacent() in Graph.py claas and tested in testIsadjacent() in UnittestDSAGraph.py	[PASSED]
Get adjacent		getadjacent() in Graph.py claas and tested in testgetadjecent() in UnittestDSAGraph.py	[PASSED]
Display As list	Display graph as list	displayAsList () in Graph.py claas and tested in test displayAsList () in UnittestDSAGraph.py	[PASSED] Displayed as list
Display as matrix	Dsplay Graph as matrix	displayAsMatrix () in Graph.py claas and tested in test displayAsMatrix () in UnittestDSAGraph.py	[PASSED] display as matrix
insert	User insert edge to the graph	insert_edge() in task1.py	[PASSED]
delete	User delete edge in graph	delete_edge() in task1.py	[PASSED]
search	User search edge	search_edge() in task1.py	[PASSED]
Read data	Read data from txt file	read_graph() in graph.py	[PASSED]
Read graph	Read graph from txt file	read_data() in graph.py	[PASSED]
Breadth first search		breadth_first_search() in graph.py	[PASSED]
Deapth first search		deapth_firs _search() in graph.py	[PASSED]
Hash Functions			
Put and get	Put and get values from the Hashtable	Put() and get () in Hash.py	[PASSED] put value and get value

		Tested in testPutAndGet() in UnitTestDSAHashtable.py	
Has key	…check values in the Hashtable….	HasKey() in Hash.py Tested in test HasKey () in UnitTestDSAHashtable.py ……	[PASSED]
remove	Remove value from the hashtable…….	Remove () in Hash.py Tested in test Remove () in UnitTestDSAHashtable.py	[PASSED]
resize	Resize hash table	Resize () in Hash.py Tested in test Resize () in UnitTestDSAHashtable.py	[PASSED]
insert	User insert data to the Hasg table	Insert() in Hashtable.py	[PASSED]
Read data	Read data from the txt file	Read_data() in Hashtable.py	[PASSED]
Read graph	Read graph from the	Read_graph() in Hashtable.py	[PASSED]
export	Export values to Hashtable	Export () in Hash.py Tested in test Export () in UnitTestDSAHashtable.py	[PASSED]
Heap functions			
Push and pop	Push and pop vales in heap	Push() and pop() in Heap.py Tested in testpushandpop() in UnitTestDSAHeap.py	[PASSED]
Is empty	Check if its empty	Is_empty() in Heap.py Tested in testisempty() in UnitTestDSAHeap.py	[PASSED]
pop on empty order	Pop value from heap	Pop() in Heap.py tested in testPopOnEmptyHeap() in UnitTestDSAHeap.py	[PASSED]
Heap order	Get priority	tested in testHeapOrder () in UnitTestDSAHeap.py	[PASSED]

Read data	Read data from text file	Read_data() in heap.py	[PASSED]
distance	Find shortest distance	dijkstra() in iter.py	[PASSED]


main
main_manu()
heap
+main()
+read_data(): int
iter
+main()
+read_data():int
+raed_graph():str
+dijkstra(0
HashTable
+main()
+insert()
+read_graph()
+read_data()
task1
+display_graph(graph)
+insert_edge()
+delete_edge()
+search_edge()
graph
+read_graph()
+read_data(0
+bfs()
+dfs()
DSAStack
- stack : DSALinkedList
+ push(Object): none
+ pop() : Object
+ peek() : Object
+ isEmpty() : boolean
+ getCount() : int
+ iterator() : Iterator
DSALinkedList
- head : DSAListNode
- tail : DSAListNode
+ CONSTRUCTOR()
+ isEmpty() : boolean
+ getCount() : int
+ insertFirst(Object)
+ insertLast(Object)
+ peekFirst() : Object
+ peekLast() : Object
+ removeFirst() : Object
+ removeLast() : Object
+ removeNode(Object)
+ hasNode(Object)
+ iterator() : Iterator
DSAListNode
- value : Object
- prev : DSAListNode
- next : DSAListNode
+ getValue() : Object
+ setValue(Object)
+ getPrev() : DSAListNode
+ setPrev(DSAListNode)
+ getNext() : DSAListNode
+ setNext(DSAListNode)
UnitTestDSAHeap
+ main(string[])
+ setup(DSAHeap)
+ testpushandpop(DSAHeap)
+ testpoponEmptyHeap(DSAHeap)
+ testIsempty(string)
+ testHeaporder(DSAHeap)
DSAHeap
- heap : DSAHeapEntry[]
- count : int
+ CONSTRUCTOR(int, object)
+ add(int, object)
+ remove() : object
+ getCount() : int
+ heapSort(Object[][]) : DSAHeapEntry[]
- trickleDown(int)
- trickleUp(int)
- getLeftChildIdx(int) : int
- getRightChildIdx(int) : int
- getParentIdx(int) : int
DSAHashtable
-hasshArr: DSAHashEntry[]
-count: int
+constructor(int)
+put(string, object0
+get(string):object
+remove(string)
+haskey(string):boolean
+export():string[]
-hash(string):int
-resize(int)
-stepHash(int):int
-nextPrime(int)
DSAGraph
- verticies : DSALinkedList
+ CONSTRUCTOR()
+ getVerticies() : DSALinkedList
+ addVertex(string)
+ addEdge(string, string)
+ getVertex(string) : DSAGraphVertex
+ hasVertex(string) : boolean
+ getValue(string) : Object
+ getVertexCount() : int
+ getEdgeCount() : int
+ getAdjacent(string) : DSALinkedList
+ isAdjacent(string, string) : boolean
+ displayAsList()
+ displayAsMatrix()
+ export(DSALinkedList, DSALinkedList)
+ iterator() : Iterator
DSAQueue
- queue : DSALinkedList
+ enqueue(Object): none
+ dequeue() : Object
+ peek() : Object
+ isEmpty() : boolean
+ getCount() : int
+ iterator() : Iterator
DSALinkedListIterator
- iterNext : DSAListNode
+ hasNext() : boolean
+ next() : Object
+ remove()
DSAHeapEntry
- priority : int
- value : object
+ CONSTRUCTOR(int, object)
DSAHashEntry
-key:string
-value:object
-state:int
-CONSTRUCTOR(string, object
UnitTestDSAGraph
+ main(string[])
+ populate(DSAGraph)
+ testHasVertex(DSAGraph)
+ testGetVertexCount(DSAGraph)
+ testGetEdgeCount(DSAGraph)
+ testGetVertex(DSAGraph)
+ testGetAdjacent(DSAGraph)
+ testIsAdjacent(DSAGraph)
+ testAddVertex(DSAGraph)
+ testAddEdge(DSAGraph)
DSAGraphVertex
- label : string
- value : Object
- links : DSALinkedList
- CONSTRUCTOR(Object)
- addEdge(DSAGraphVertex)
UnitTestDSALinkedList
+ main(string[])
+ title(string)
+ testRemoval(DSALinkedList)
+ testInsertion(DSALinkedList)
+ testEmpty(DSALinkedList)
UnitTestDSAHahtable
+ main(string[])
+testPutandGest(DSAHash
+testHaskey(DSAHash)
+testremove(DSAHash)
+testresize(DSAHash)
+testexport(DSAHash)
UnitTestDSAStack
+ main(string[])
UnitTestDSAQueue
+ main(string[])
maintest
+main_menu([])