Contents

Introduction 2
Class diagram 2
Activity Diagram 3
System sequence diagram 4
Data Structures and Algorithms Used 4
Graph Data Structure: 4
Breadth-First Search (BFS): 4
Depth-First Search (DFS): 4
Hash Table: 4
Priority Queue: 5
Description of classes 5
Graph: 5
• Description 5
• Methods 5
LocationHashTable: 5
Description: 5
Methods: 5
Itinerary: 5
• Description 5
• Methods 5
Location: 5
Description: 5
Methods: 6
2. Menu 6
Description: 6
Methods: 6
Testing Methodology 7
Implementation Reflection and Improvements: 9
Conclusion 10

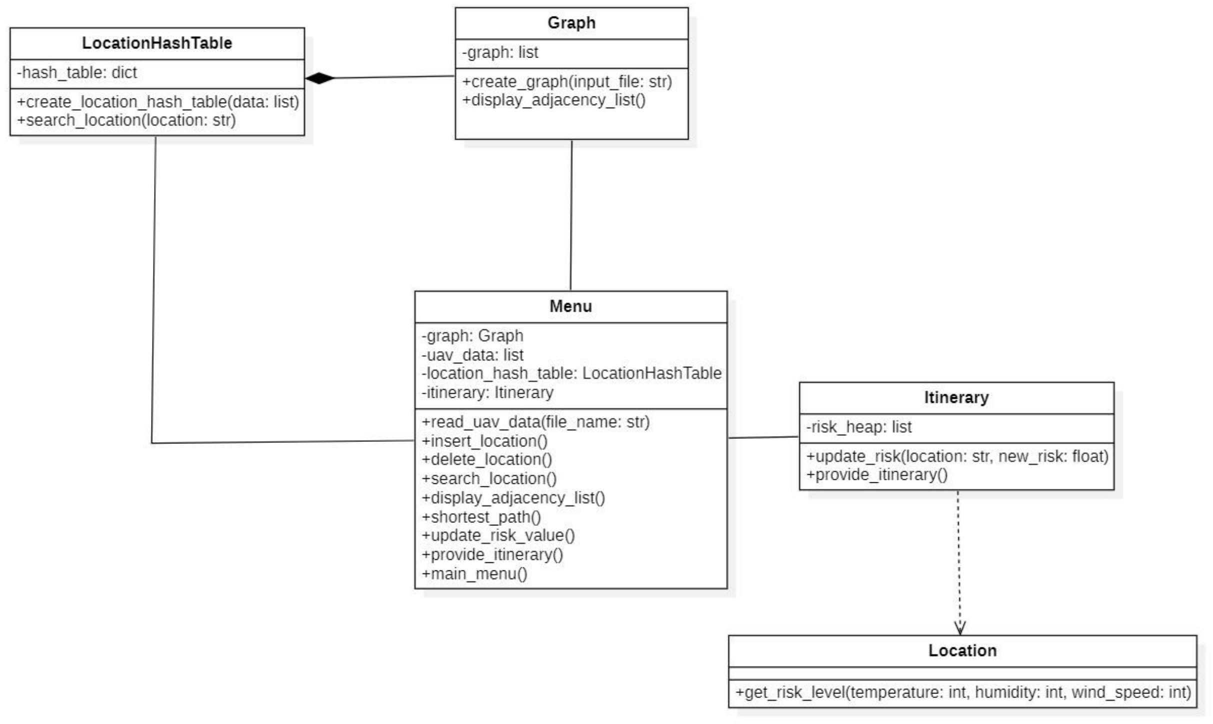
References 10

Introduction

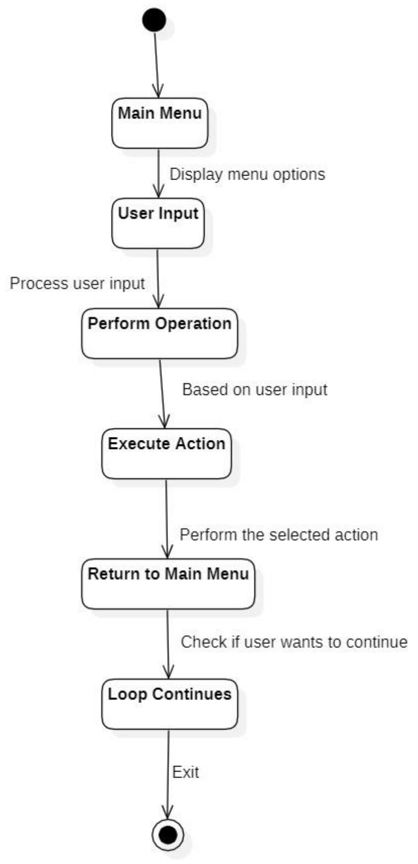
This code includes a software that manages a graph representing locations and their connections. It allows customers to perform numerous operations at the graph, together with putting a location and side, deleting a location, trying to find a place, showing the adjacency listing, locating the shortest course among places, updating the risk value of a vicinity, and imparting an itinerary based on the threat values. The principal characteristic serves because the access point of the program. It creates the graph from an enter file, reads UAV statistics from any other record, and initializes a threat heap. It then presents a menu to the consumer, letting them pick out unique operations to carry out on the graph. The create_graph feature reads a document containing the graph facts and creates an adjacency listing illustration of the graph. The display_adjacency_list feature prints the adjacency listing of the graph. The application also consists of a take a look at magnificence TestGraph that defines unit checks for the create_graph and display_adjacency_list features.

Uml Diagrams

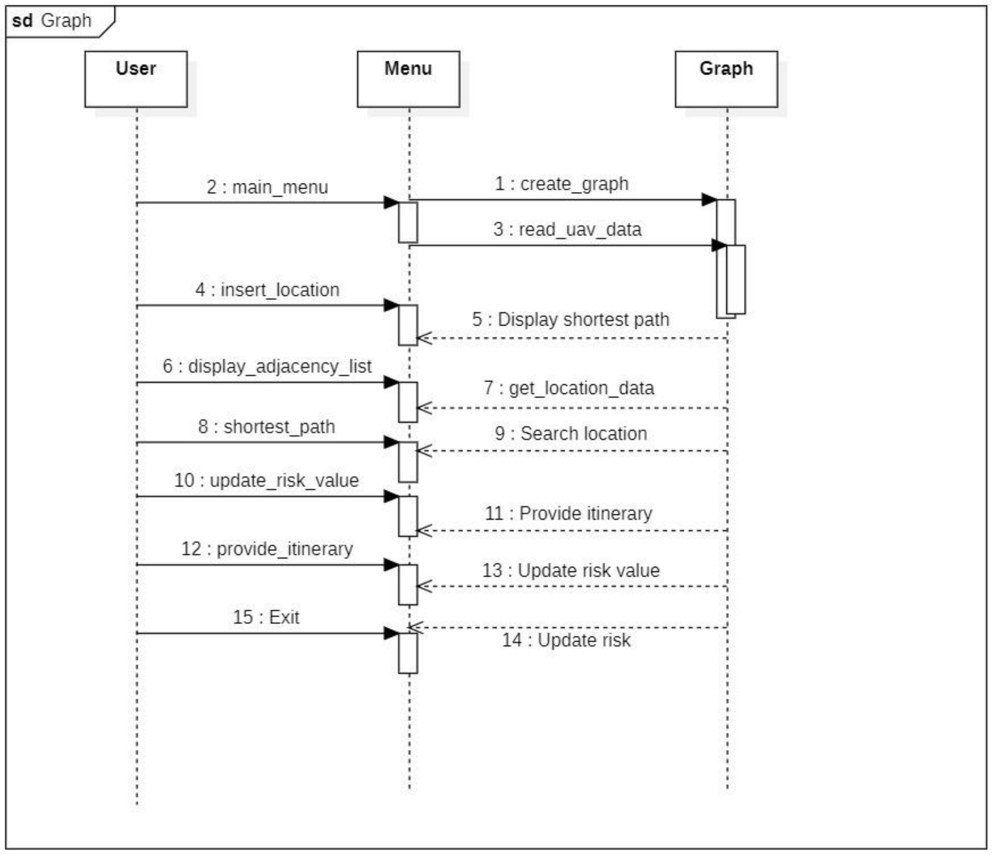
Class diagram


LocationHashTable
-hash_table: dict
+create_location_hash_table(data: list)
+search_location(location: str)
Graph
-graph: list
+create_graph(input_file: str)
+display_adjacency_list()
Menu
-graph: Graph
-uav_data: list
-location_hash_table: LocationHashTable
-itinerary: Itinerary
+read_uav_data(file_name: str)
+insert_location()
+delete_location()
+search_location()
+display_adjacency_list()
+shortest_path()
+update_risk_value()
+provide_itinerary()
+main_menu()
Itinerary
-risk_heap: list
+update_risk(location: str, new_risk: float)
+provide_itinerary()
Location
+get_risk_level(temperature: int, humidity: int, wind_speed: int)

Activity Diagram



System sequence diagram



Data Structures and Algorithms Used

Following data structures are used in our program

Graph Data Structure:

The code represents a graph using an adjacency listing. The graph is stored as a listing of lists, wherein each internal listing represents the associates of a vertex.

Breadth-First Search (BFS):

The bfs characteristic implements the BFS algorithm to find the shortest route between vertices inside the graph. It makes use of a queue (applied the use of deque) to carry out the breadth-first traversal.

Depth-First Search (DFS):

The dfs feature implements the DFS set of rules to traverse the graph beginning from a given vertex. It uses recursion to go to all linked vertices.

Hash Table:

The LocationHashTable class represents a hash table that stores area statistics. It uses a dictionary (hash_table) to map locations to their corresponding information.

Priority Queue:

The Itinerary elegance uses a priority queue (implemented as a heap the use of heapq) to keep locations and their related hazard values. The priority queue is used to provide an itinerary taken care of by way of hazard fee.

Description of classes

Graph:

- Description:

  Represents a graph the usage of an adjacency listing.

- Methods:

  - **create_graph(input_file)**: Creates the graph by using reading information from an input document and populating the adjacency listing.
  - **display_adjacency_list()**: Displays the adjacency list representation of the graph.

LocationHashTable:

Description:

Stores location facts the usage of a hash desk.

Methods:

- **create_location_hash_table(data)**: Creates the hash table by parsing and storing location records.
- **search_location(location)**: Searches for a area within the hash table and returns its associates.

Itinerary:

- Description:

  Manages a concern queue of places and their associated risk values.

- Methods:

  - **update_risk(location, new_risk)**: Updates the threat value for a place in the priority queue.
  - **provide_itinerary()**: Provides an itinerary by using returning the places in the precedence queue sorted by means of chance value.

Location:

Description:

Provides software techniques associated with region hazard degrees.

Methods:

- **get_risk_level(temperature, humidity, wind_speed)**: Calculates the chance degrees for temperature, humidity, and wind velocity values and returns the best hazard level

2. ## Menu:

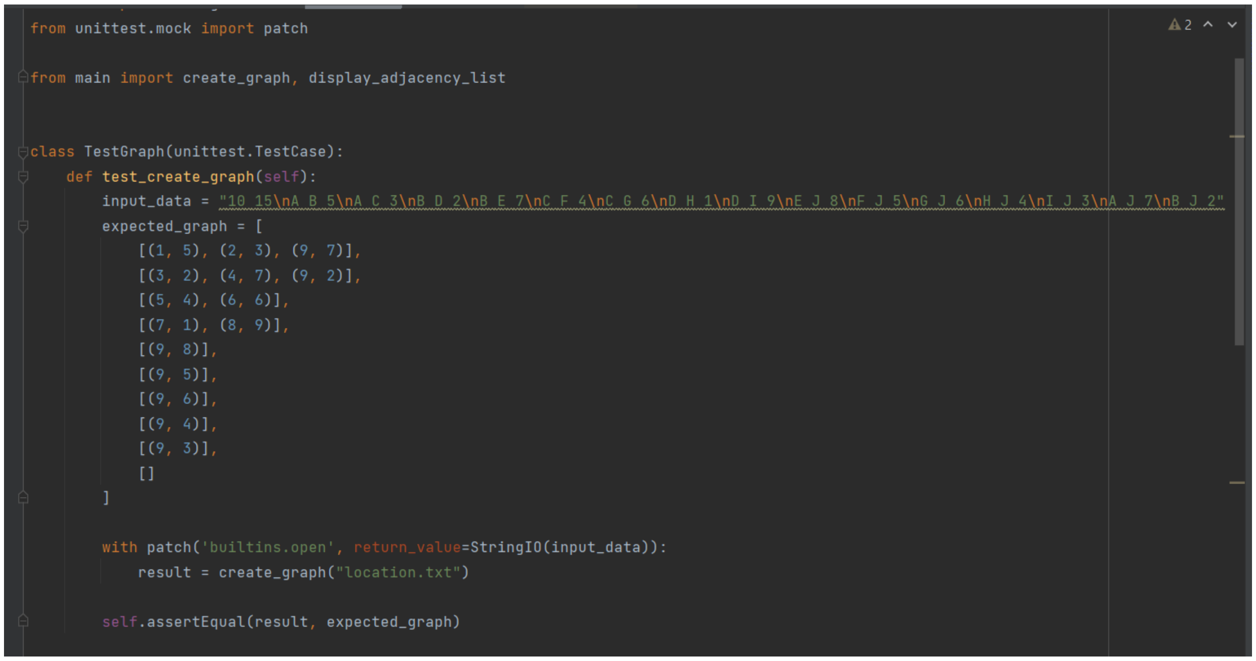
Description:

Represents the principle menu for interacting with the program.

Methods:

- **read_uav_data(file_name)**: Reads UAV (Unmanned Aerial Vehicle) data from a file and stores it.
- **insert_location()**: Inserts a location and an edge into the graph.
- **delete_location()**: Deletes a location from the graph.
- **search_location()**: Searches for a location and displays its neighbors.
- **display_adjacency_list()**: Displays the adjacency list illustration of the graph..
- **shortest_path()**: Calculates and shows the shortest path between two locations within the graph.
- **update_risk_value()**: Updates the risk value for a location in the itinerary.
- **provide_itinerary()**: Displays the itinerary of locations sorted via chance cost.
- **main_menu()**: Displays the primary menu and handles person enter.

Testing Methodology

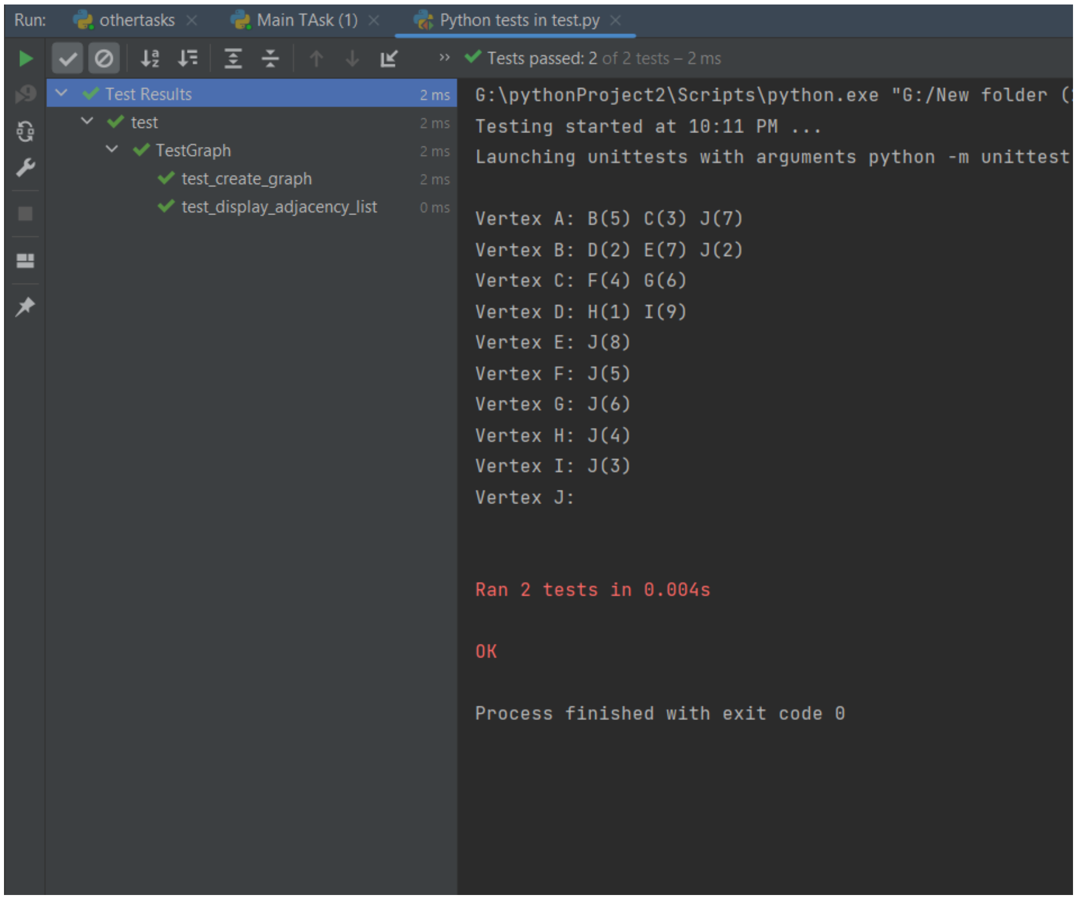


```python
from unittest.mock import patch

from main import create_graph, display_adjacency_list


class TestGraph(unittest.TestCase):
    def test_create_graph(self):
        input_data = "10 15\nA B 5\nA C 3\nB D 2\nB E 7\nC F 4\nC G 6\nD H 1\nD I 9\nE J 8\nF J 5\nG J 6\nH J 4\nI J 3\nA J 7\nB J 2"
        expected_graph = [
            [(1, 5), (2, 3), (9, 7)],
            [(3, 2), (4, 7), (9, 2)],
            [(5, 4), (6, 6)],
            [(7, 1), (8, 9)],
            [(9, 8)],
            [(9, 5)],
            [(9, 6)],
            [(9, 4)],
            [(9, 3)],
            []
        ]

        with patch('builtins.open', return_value=StringIO(input_data)):
            result = create_graph("location.txt")

        self.assertEqual(result, expected_graph)
```

1. **test_create_graph**:
   - This test verifies the correctness of the create_graph function, which is responsible for creating a graph from input data.
   - The test defines an input string (input_data) representing the graph data and an expected graph representation (expected_graph).
   - By patching the open function to return a StringIO object containing the input data, the test calls the create_graph function with a dummy file name.
   - The result is compared against the expected graph using the assertEqual method.
   - If the result matches the expected graph, the test passes.
2. **test_display_adjacency_list**:
   - This test verifies the correctness of the display_adjacency_list function, which is responsible for displaying the adjacency list representation of the graph.
   - The test defines a graph (graph) and an expected output string (expected_output) representing the adjacency list representation.
   - By redirecting the standard output to a StringIO object, the test calls the display_adjacency_list function with the graph.
   - The captured output is compared against the expected output using the assertEqual method.
   - If the captured output matches the expected output, the test passes.

The test results show that both tests have passed successfully:

- The first test ran **create_graph** with the provided input data and verified that the returned graph matches the expected graph representation.
- The second test ran **display_adjacency_list** with the provided graph and verified that the output matches the expected adjacency list representation.



```
Test Results                    2 ms
  test                          2 ms
    TestGraph                   2 ms
      test_create_graph         2 ms
      test_display_adjacency_list  0 ms
```

```
G:\pythonProject2\Scripts\python.exe "G:/New folder (
Testing started at 10:11 PM ...
Launching unittests with arguments python -m unittest

Vertex A: B(5) C(3) J(7)
Vertex B: D(2) E(7) J(2)
Vertex C: F(4) G(6)
Vertex D: H(1) I(9)
Vertex E: J(8)
Vertex F: J(5)
Vertex G: J(6)
Vertex H: J(4)
Vertex I: J(3)
Vertex J:


Ran 2 tests in 0.004s

OK

Process finished with exit code 0
```

Implementation Reflection and Improvements:

Reflecting on our implementation, there are numerous regions where we can make improvements to enhance the code and consumer enjoy.

Firstly, we should recollect organizing the code in a more based way. By isolating the training into separate files or modules, we are able to enhance maintainability and readability.

it would be useful to follow regular naming conventions at some stage in the codebase. Using a unmarried conference for training, techniques, and variables, inclusive of either snake_case or camelCase, will make the code greater cohesive.

To provide a higher user revel in, we have to enforce input validation and mistakes managing. By validating user inputs for correct format and variety, we will save you surprising behavior and provide useful error messages.

It is crucial to validate the input records from files earlier than processing it. Verifying that the information suits the anticipated layout will help avoid errors at some stage in runtime.

Separating the I/O operations from the core common sense of the instructions could enhance modularity and testability. By moving file reading and writing operations to separate utility features or instructions, we will make the code more prepared and maintainable.

To make sure the correctness of the implementation, we have to amplify the unit check insurance. Including extra eventualities and facet instances in our checks will assist identify and fasten potential troubles.

Error handling must be added throughout the code to deal with surprising conditions gracefully. By presenting informative blunders messages, we are able to manual users whilst something goes wrong.

Identifying possibilities for code reusability will assist remove redundancy. Creating software functions or classes for commonplace operations or calculations can decorate code reuse and improve performance.

Consistent code formatting is essential for readability. Following a fashion guide or the use of a code formatter tool will make certain uniform formatting across the codebase.

If this system is meant for real usage, we must don't forget enhancing the user interface. Implementing a graphical person interface (GUI) library or a web-based totally interface can enhance the overall user revel in.

Lastly, relying on the dimensions and complexity of the graph, we may additionally need to optimize the algorithms used for calculating the shortest direction or updating hazard values. This will assist improve overall performance if these operations come to be overall performance bottlenecks

Conclusion

Our implementation of the code has supplied the muse for the favored functionality, but there are numerous regions that may be progressed to decorate the overall excellent of the code. By organizing the code into separate files or modules and following consistent naming conventions, we will enhance the code's structure and clarity. Implementing enter validation and blunders coping with will make the code extra strong and prevent sudden conduct. Verifying the input statistics and setting apart I/O operations from the middle good judgment will improve statistics integrity and modularity. Expanding the unit check insurance, imposing mistakes coping with in the course of the code, and including documentation will decorate the code's correctness, reliability, and maintainability. Identifying opportunities for code reusability and applying regular code formatting will improve performance and clarity. Considering the implementation of a graphical person interface (GUI) or internet-primarily based interface can beautify the user revel in and make the program extra person-pleasant. Finally, optimizing algorithms for calculating the shortest direction or updating threat values can be essential if the graph length and complexity increase substantially to ensure gold standard performance. By addressing those upgrades and tailoring them to particular venture requirements, we will in addition refine and optimize our implementation.

References

“heapq - Heap queue algorithm." Python Documentation. Available online: https://docs.python.org/3/library/heapq.html

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.