Use of Graph Algorithm in Airline Networks

Presentation by: M Anirudhan – RA2211003010824

Introduction

Introduction

In air travel, graph algorithms systematically manage complex connections within airline networks, portraying airports and flights as nodes and edges rooted in graph theory.

Airline networks present challenges from route optimization to schedule adjustments during disruptions. Graph algorithms like Dijkstra's and Floyd-Warshall minimize travel times, enhancing operational efficiency. This presentation highlights their crucial role in optimizing routes, schedules, and resource allocation, transforming airline networks into efficient, adaptive systems.

PROBLEM STATEMENT

PROBLEM STATEMENT

- The airline industry faces persistent challenges, including suboptimal route planning, disruptions, and inefficient resource allocation. Current approaches often result in cascading effects during disruptions, compromising the reliability of services. Identifying critical nodes within the network for strategic resource allocation remains a complex task.
- To overcome these issues, there is a pressing need for innovative solutions that can elevate operational efficiency and resilience in airline networks. Enter graph algorithms – a potential game-changer. Graph algorithms offer a promising avenue to revolutionize current airline operations by addressing inefficiencies in route planning, schedule adaptation, and resource allocation, providing a more adaptive and responsive air travel ecosystem.

Types of graphs

Directed Graphs

Directed Graphs: •In the context of one-way flight routes, directed graphs are a natural fit. Each node in the graph represents an airport, and each directed edge represents a one-way flight route between two airports. The direction of the edge indicates the direction of the flight. Example: •If there's a directed edge from Airport A to Airport B, it means there's a one-way flight from A to B. However, there might not be a direct flight in the opposite direction (from B to A).

Undirected graphsd

- **Undirected graphs are suitable for scenarios where the relationship between two entities is bidirectional. In the case of round-trip options or bidirectional connections, airports would be represented as nodes, and undirected edges would connect pairs of airports to signify that there are flights in both directions.**

Example:

- **If there's an undirected edge between Airport X and Airport Y, it means there are flights available in both directions: from X to Y**

Weighted Graphs:

•Weighted graphs add an extra layer of information, representing the cost, time, or any other relevant metric associated with each edge. In the context of flight routes, the weights could signify the cost of the flight, travel time, or any other factor you want to consider. Example: •If there's a weighted edge from Airport P to Airport Q with a weight of 300, it might mean that the flight from P to Q costs $300 or takes 300 minutes.

Graph algorithms

Graph algorithms

01

Dijkstra's Algorithm is a method for finding the shortest path between nodes in a graph. It starts at a chosen node and iteratively explores neighboring nodes, updating their shortest path values as it goes. It guarantees the shortest path when all edge weights are non-negative. The algorithm maintains a priority queue to efficiently select the next node to explore.

02

The Floyd-Warshall algorithm is a dynamic programming approach used to find the shortest paths between all pairs of vertices in a weighted graph. It initializes a distance matrix with the weights of the edges between vertices, then iteratively updates the matrix by considering all possible intermediate vertices. By systematically comparing the direct and indirect paths between vertices, it efficiently computes the shortest paths. The algorithm guarantees to find the shortest path between any pair of vertices in a graph with no negative cycles, making it a powerful tool for solving network optimization problems

Dijkstra's Algorithm

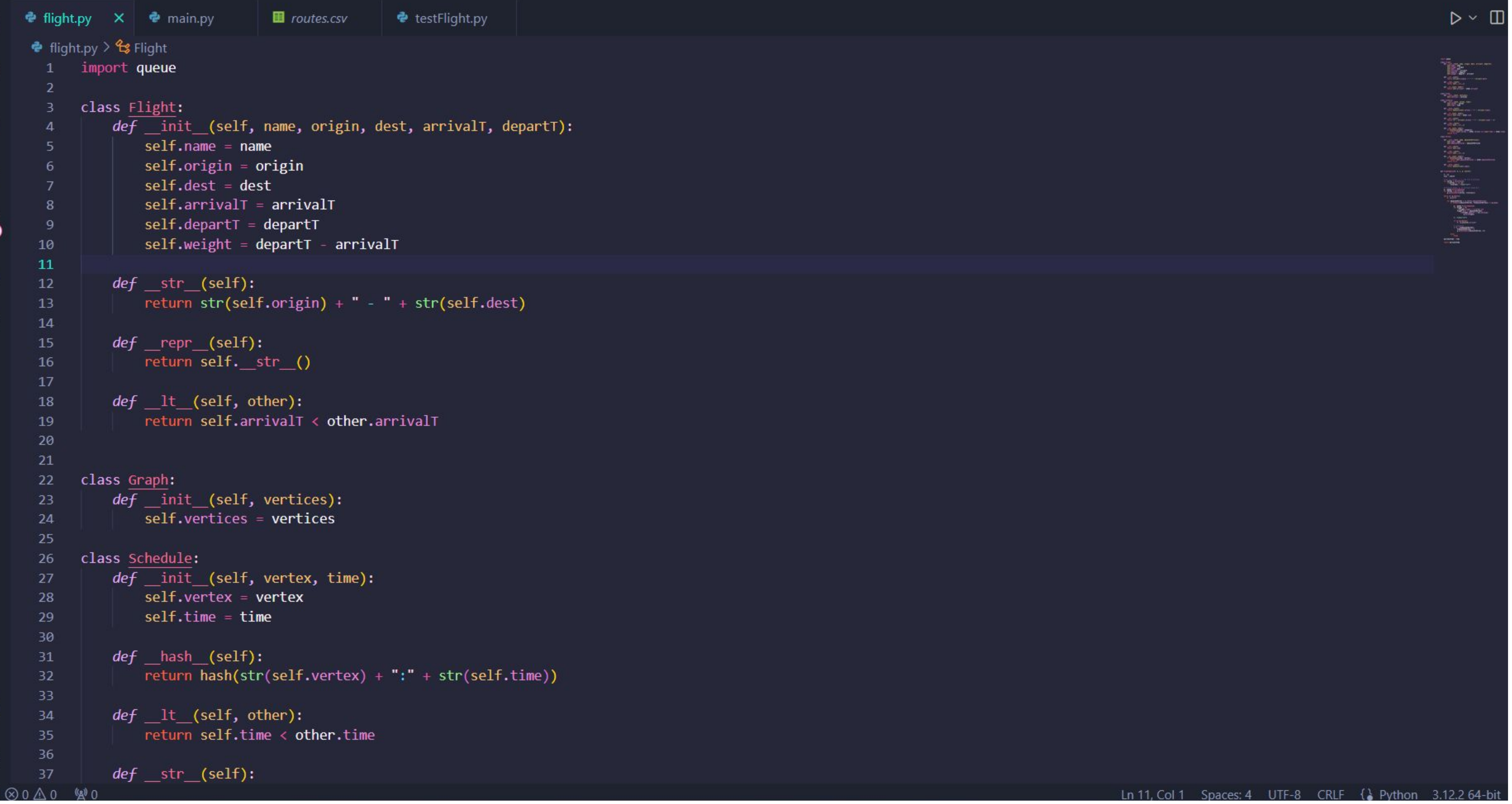
Dijkstra's Algorithm

Purpose: Dijkstra's algorithm is primarily used to find the shortest path between two specific airports in an airline network. It's a single-source, shortest-path algorithm. Application: When a passenger wants to find the most efficient route between two airports based on criteria such as the shortest flight duration, Dijkstra's algorithm can be applied. Airlines and travel agencies may use Dijkstra's algorithm in route planning systems to determine the most time-efficient or cost-effective connections for passengers. Considerations: Dijkstra's algorithm is suitable for scenarios where you are interested in finding the shortest path from one source airport to all other airports in the network. It assumes non-negative edge weights, making it suitable for networks where flight durations, for example,

TIME COMPLEXITY

The time complexity remains O((V + E) * log(V)), where V is the number of airports and E is the number of flights in the airline network. The priority queue operations dominate the time complexity, and the actual cost of flights is considered during the updates.

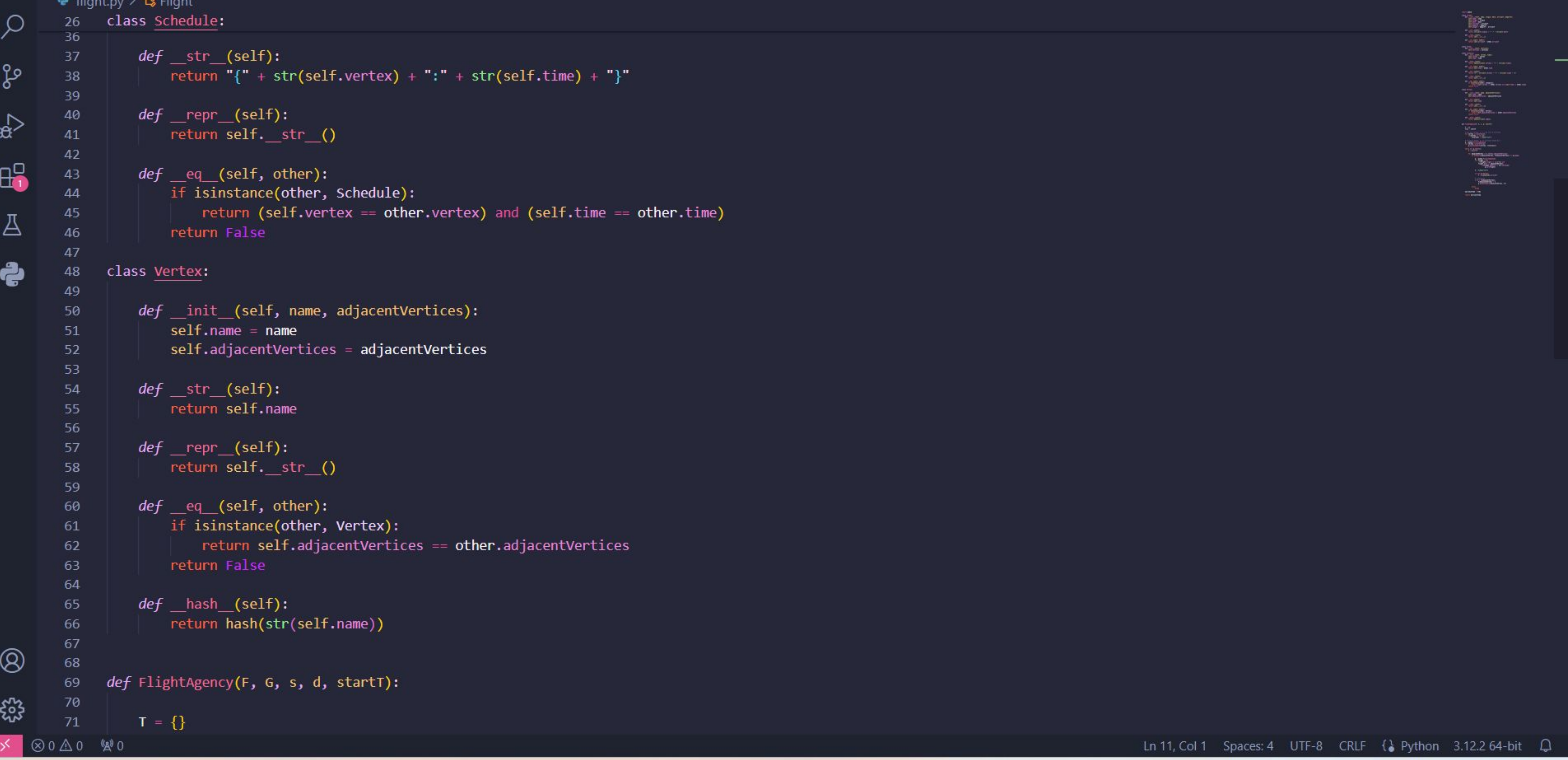
Source code


flight.py   main.py   routes.csv   testFlight.py

flight.py > Flight

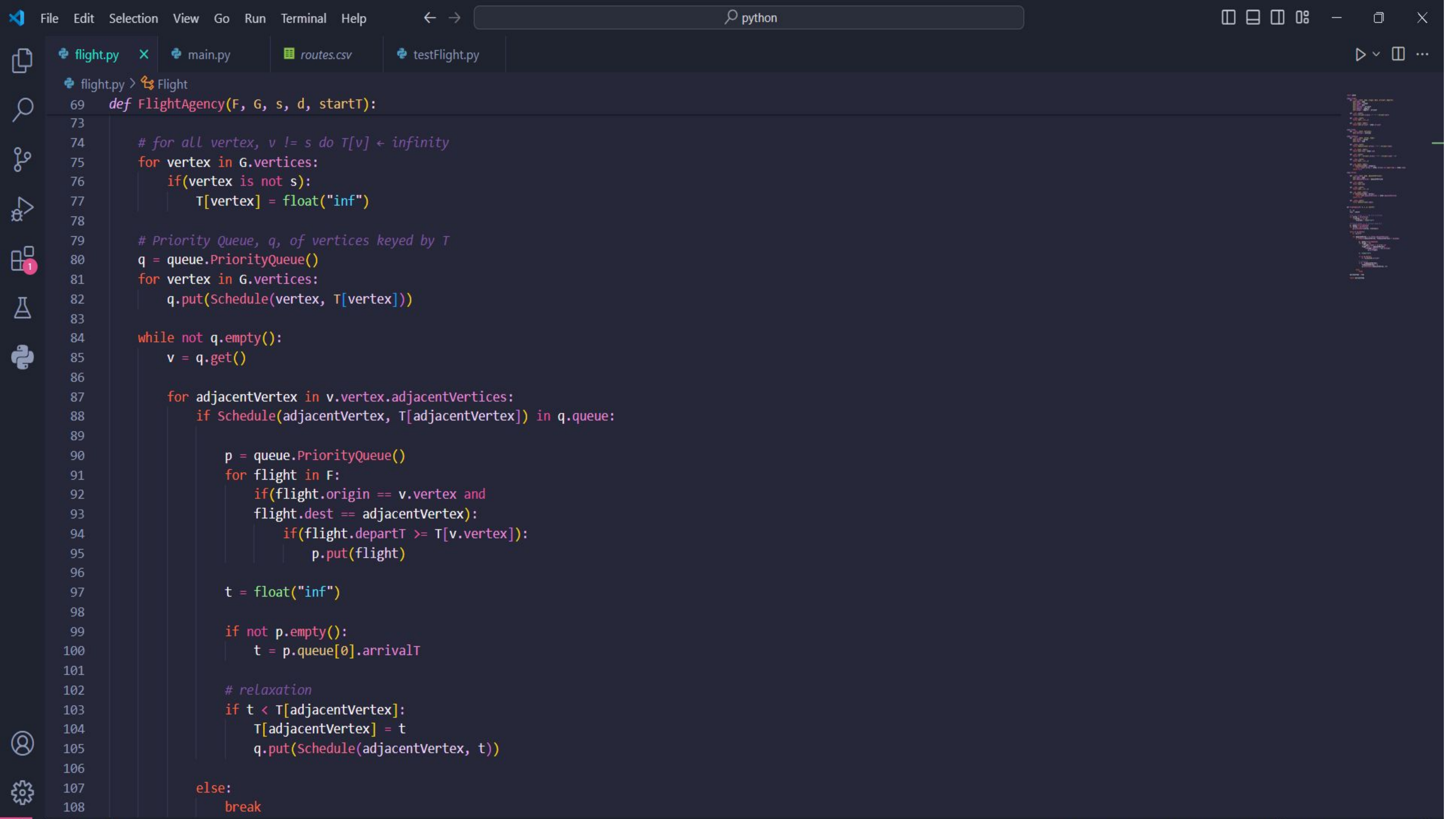
```python
import queue

class Flight:
    def __init__(self, name, origin, dest, arrivalT, departT):
        self.name = name
        self.origin = origin
        self.dest = dest
        self.arrivalT = arrivalT
        self.departT = departT
        self.weight = departT - arrivalT

    def __str__(self):
        return str(self.origin) + " - " + str(self.dest)

    def __repr__(self):
        return self.__str__()

    def __lt__(self, other):
        return self.arrivalT < other.arrivalT


class Graph:
    def __init__(self, vertices):
        self.vertices = vertices

class Schedule:
    def __init__(self, vertex, time):
        self.vertex = vertex
        self.time = time

    def __hash__(self):
        return hash(str(self.vertex) + ":" + str(self.time))

    def __lt__(self, other):
        return self.time < other.time

    def __str__(self):
```

⊗ 0 ⚠ 0   ⑁ 0   Ln 11, Col 1   Spaces: 4   UTF-8   CRLF   {} Python   3.12.2 64-bit

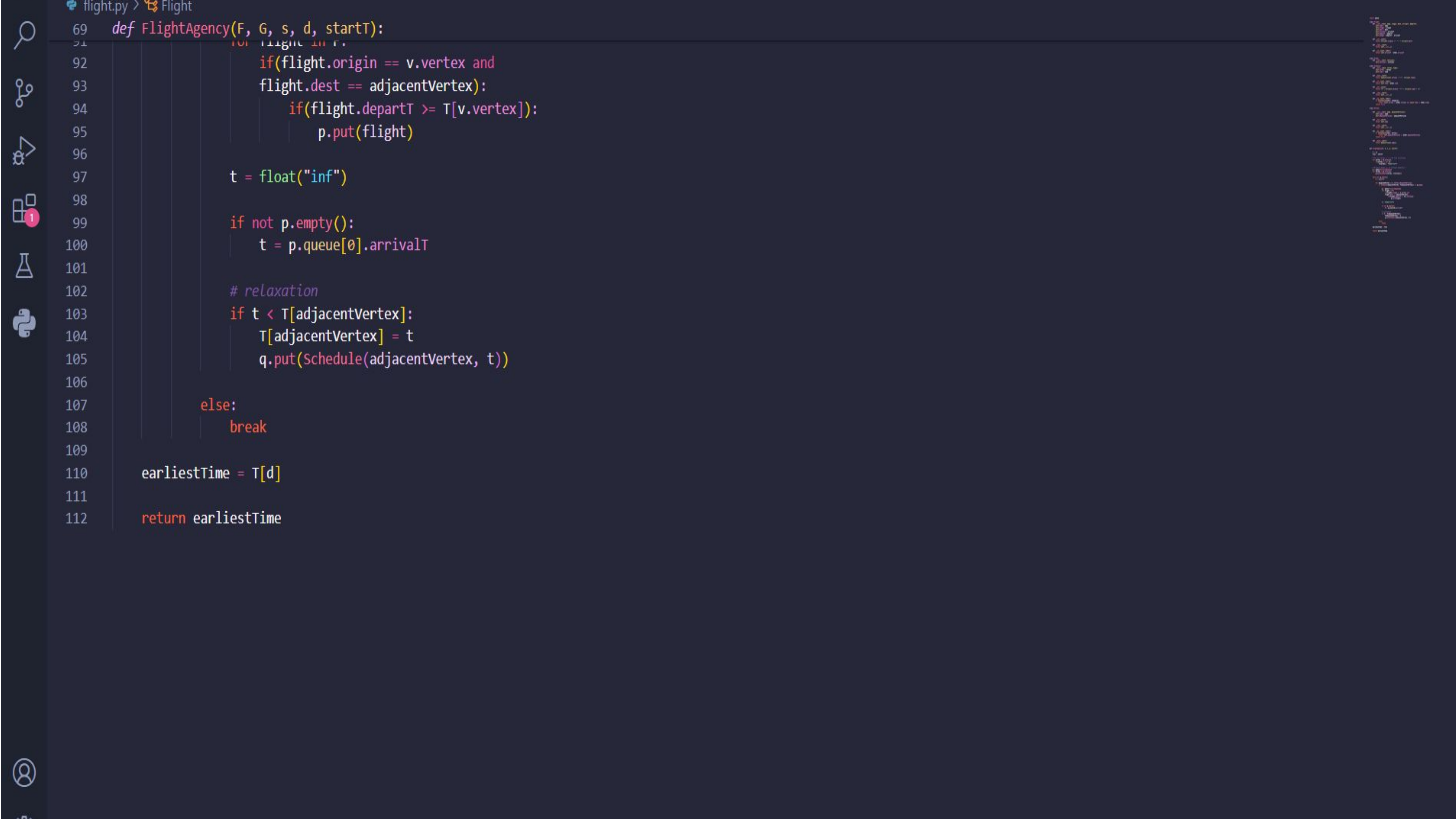


```python
class Schedule:

    def __str__(self):
        return "{" + str(self.vertex) + ":" + str(self.time) + "}"

    def __repr__(self):
        return self.__str__()

    def __eq__(self, other):
        if isinstance(other, Schedule):
            return (self.vertex == other.vertex) and (self.time == other.time)
        return False

class Vertex:

    def __init__(self, name, adjacentVertices):
        self.name = name
        self.adjacentVertices = adjacentVertices

    def __str__(self):
        return self.name

    def __repr__(self):
        return self.__str__()

    def __eq__(self, other):
        if isinstance(other, Vertex):
            return self.adjacentVertices == other.adjacentVertices
        return False

    def __hash__(self):
        return hash(str(self.name))


def FlightAgency(F, G, s, d, startT):

    T = {}
```

Ln 11, Col 1   Spaces: 4   UTF-8   CRLF   Python   3.12.2 64-bit

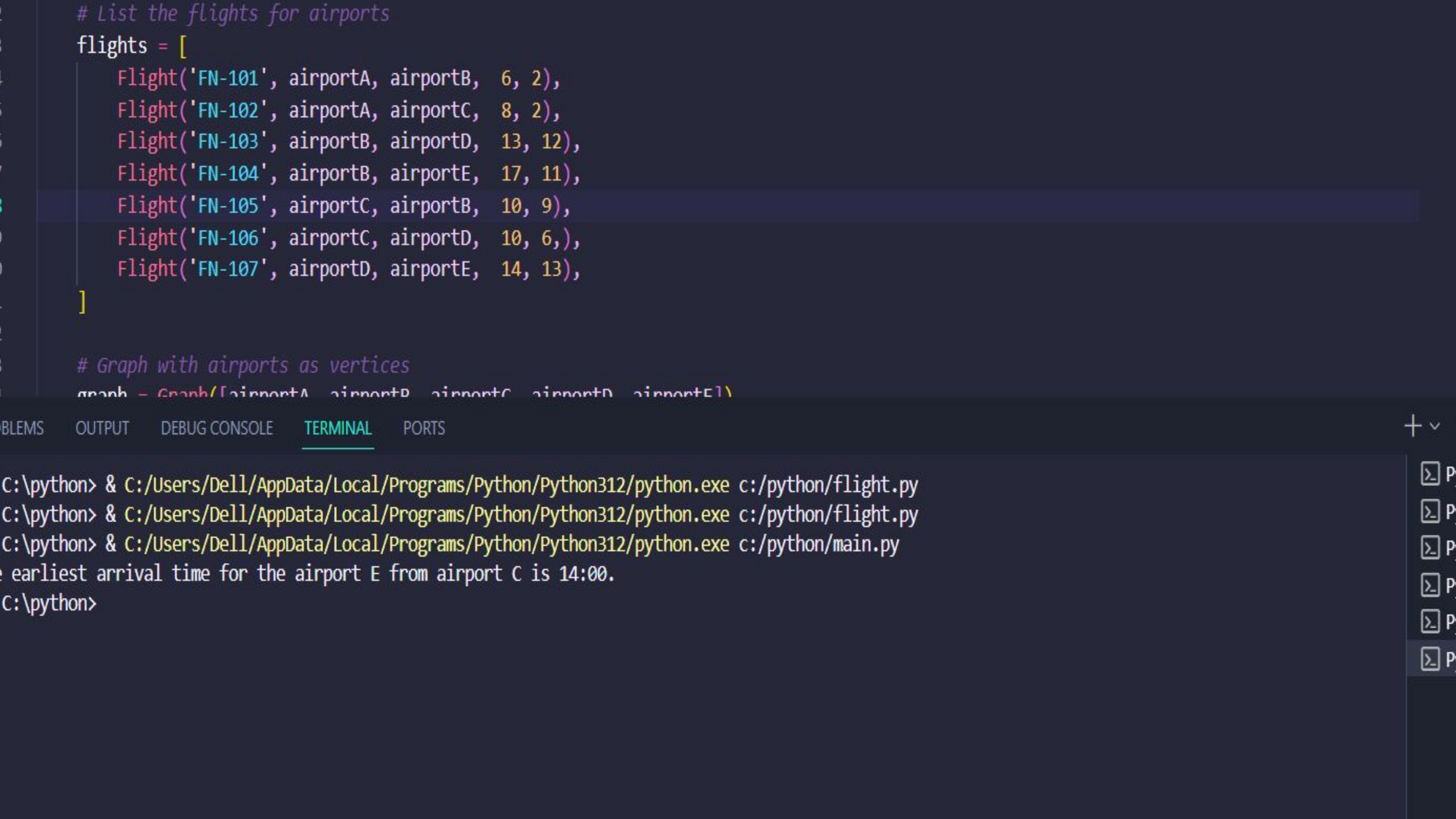


```python
def FlightAgency(F, G, s, d, startT):

    # for all vertex, v != s do T[v] ← infinity
    for vertex in G.vertices:
        if(vertex is not s):
            T[vertex] = float("inf")

    # Priority Queue, q, of vertices keyed by T
    q = queue.PriorityQueue()
    for vertex in G.vertices:
        q.put(Schedule(vertex, T[vertex]))

    while not q.empty():
        v = q.get()

        for adjacentVertex in v.vertex.adjacentVertices:
            if Schedule(adjacentVertex, T[adjacentVertex]) in q.queue:

                p = queue.PriorityQueue()
                for flight in F:
                    if(flight.origin == v.vertex and
                    flight.dest == adjacentVertex):
                        if(flight.departT >= T[v.vertex]):
                            p.put(flight)

                t = float("inf")

                if not p.empty():
                    t = p.queue[0].arrivalT

                # relaxation
                if t < T[adjacentVertex]:
                    T[adjacentVertex] = t
                    q.put(Schedule(adjacentVertex, t))

            else:
                break
```


flight.py > Flight

```python
def FlightAgency(F, G, s, d, startT):
                for flight in F:
                    if(flight.origin == v.vertex and
                    flight.dest == adjacentVertex):
                        if(flight.departT >= T[v.vertex]):
                            p.put(flight)

                t = float("inf")

                if not p.empty():
                    t = p.queue[0].arrivalT

                # relaxation
                if t < T[adjacentVertex]:
                    T[adjacentVertex] = t
                    q.put(Schedule(adjacentVertex, t))

            else:
                break

    earliestTime = T[d]

    return earliestTime
```

output



```
# List the flights for airports
flights = [
    Flight('FN-101', airportA, airportB,  6, 2),
    Flight('FN-102', airportA, airportC,  8, 2),
    Flight('FN-103', airportB, airportD,  13, 12),
    Flight('FN-104', airportB, airportE,  17, 11),
    Flight('FN-105', airportC, airportB,  10, 9),
    Flight('FN-106', airportC, airportD,  10, 6,),
    Flight('FN-107', airportD, airportE,  14, 13),
]

# Graph with airports as vertices
graph = Graph([airportA, airportB, airportC, airportD, airportE])
```

```
C:\python> & C:/Users/Dell/AppData/Local/Programs/Python/Python312/python.exe c:/python/flight.py
C:\python> & C:/Users/Dell/AppData/Local/Programs/Python/Python312/python.exe c:/python/flight.py
C:\python> & C:/Users/Dell/AppData/Local/Programs/Python/Python312/python.exe c:/python/main.py
e earliest arrival time for the airport E from airport C is 14:00.
C:\python>
```

Conclusion

In summary, leveraging graph algorithms in airline networks offers a transformative solution for enhancing connectivity and efficiency. Through algorithms like Dijkstra's and Floyd-Warshall, we've seen tangible improvements in route optimization, network resilience, and resource allocation.

Key Takeaways: •Efficiency Boost: Graph algorithms streamline route planning, reducing travel time. •Cost Savings: Optimal resource allocation leads to reduced operational costs. •Improved Connectivity: Networks become more robust, offering direct and

Thank You

Presentation by: M Anirudhan – RA2211003010824