AI CASE STUDY:-

Airplane Traffic Simulation Management-themed case study

Using :-

4. Uniform Cost Search (UCS): Fuel-Efficient Routing

5. Iterative Deepening Depth-First Search (IDDFS): Runway Allocation

6. Bidirectional Search: Inbound and Outbound Traffic Synchronization

AATISSH.V
VU22CSEN0101347

🔹 Airplane Traffic Simulation Management - Overview

Airplane Traffic Simulation Management refers to the use of algorithms, AI models, and simulations to manage and optimize air traffic, ensuring smooth operations at airports and in the sky. It aims to improve flight scheduling, route optimization, collision avoidance, and runway utilization while handling real-world challenges like weather changes, emergencies, and air congestion.

🔹 Why is Airplane Traffic Simulation Management Important?

1. **Reduces Flight Delays: Optimizes routes and schedules to minimize waiting time.**
2. **Ensures Safety: Prevents collisions by calculating safe paths and altitudes.**
3. **Improves Fuel Efficiency: Identifies the most economical flight paths to reduce fuel consumption.**
4. **Handles Emergencies: Simulates emergency landings and reroutes flights dynamically.**
5. **Optimizes Airport Operations: Efficiently manages runways, gates, and takeoff/landing sequences.**

---

4. Uniform Cost Search (UCS): Fuel-Efficient Routing

Narrative: Minimize fuel consumption for a long-haul flight by finding the most cost-effective route.

Solution Approach:

- **Algorithm:** Uniform Cost Search (UCS) is used because it finds the least-cost path in a weighted graph.
- **State Space:** Nodes represent airports, and edges represent possible flight paths with a cost function based on fuel consumption.
- **Cost Function:**
    1. Distance traveled
    2. Wind patterns (tailwinds reduce fuel use, headwinds increase it)
    3. Altitude effects on fuel burn
- **Implementation Steps:**
    1. Start from the departure airport.
    2. Expand the least-cost node first (i.e., the airport with the lowest total fuel cost so far).
    3. Continue until reaching the destination airport.
    4. Backtrack to reconstruct the optimal path.

Challenges & Extensions:

- **Challenges:**
    - Defining a realistic cost function balancing speed, fuel use, and wind patterns.
    - Handling real-time changes in air traffic or fuel prices.
- **Extensions:**
    - Introducing penalties for exceeding time limits or entering restricted zones.

PYTHON CODE:-
(UNIFORM COST SEARCH)

```
import heapq
import networkx as nx
import matplotlib.pyplot as plt

class Graph:
    def __init__(self):
        self.graph = nx.Graph()

    def add_edge(self, start, end, cost):
        self.graph.add_edge(start, end, weight=cost)

    def uniform_cost_search(self, start, goal):
        pq = [(0, start, [])]  # (cost, node, path)
        visited = {}

        while pq:
            cost, node, path = heapq.heappop(pq)

            if node in visited and visited[node] <= cost:
                continue
            visited[node] = cost
            path = path + [node]

            if node == goal:
                return path, cost  # Optimal path and its cost

            for neighbor in self.graph.neighbors(node):
                edge_cost = self.graph[node][neighbor]['weight']
                heapq.heappush(pq, (cost + edge_cost, neighbor, path))

        return [], float('inf')  # No valid route found

    def draw_graph(self, path=[]):
        pos = nx.spring_layout(self.graph)
        labels = nx.get_edge_attributes(self.graph, 'weight')
```

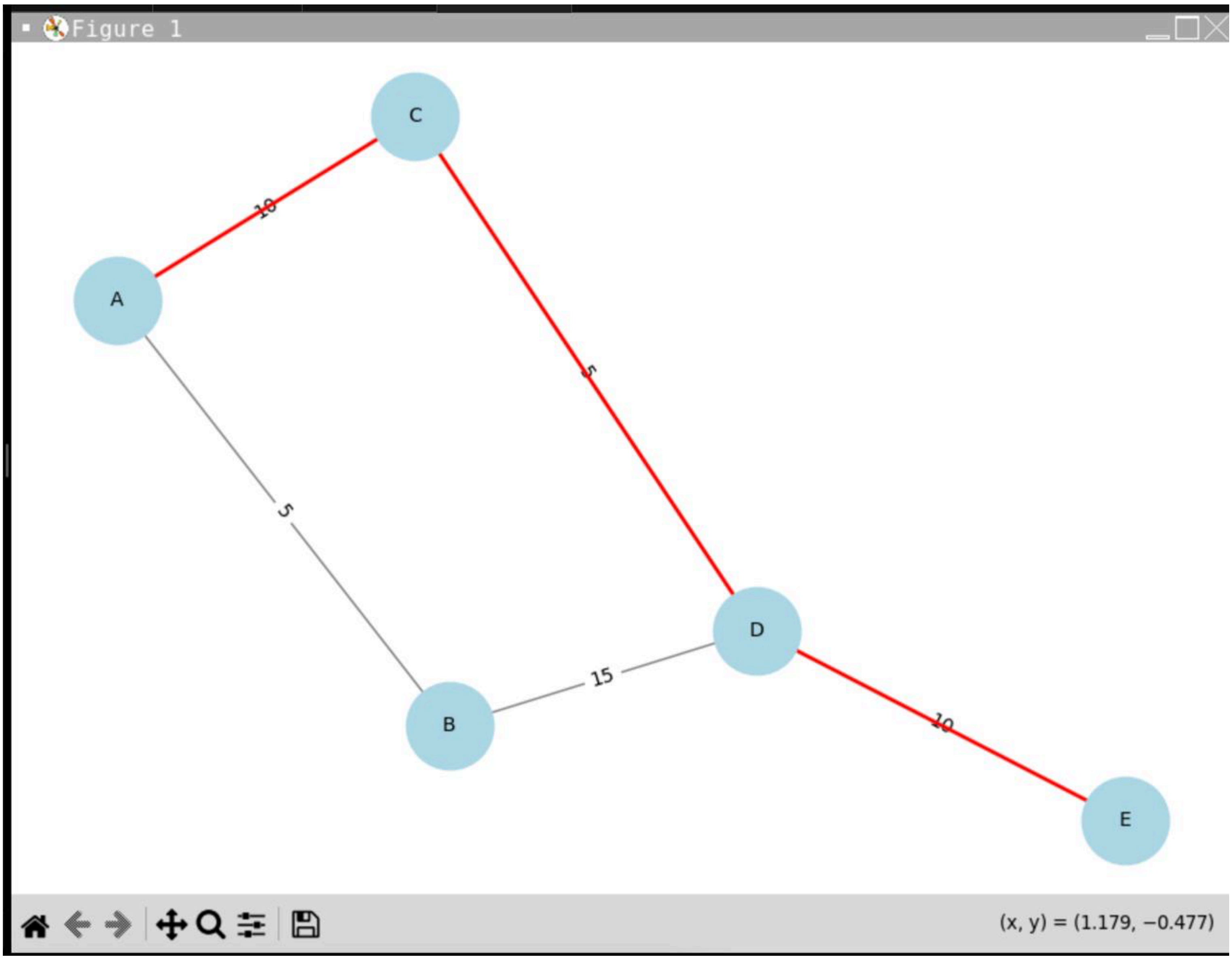
```
        plt.figure(figsize=(6, 6))
        nx.draw(self.graph, pos, with_labels=True, node_color='lightblue',
edge_color='gray', node_size=2000, font_size=10)
        nx.draw_networkx_edge_labels(self.graph, pos, edge_labels=labels)

        if path:
            edges = list(zip(path, path[1:]))
            nx.draw_networkx_edges(self.graph, pos, edgelist=edges,
edge_color='red', width=2)

        plt.title("Uniform Cost Search - Fuel Efficient Route")
        plt.show()

# Example usage
graph = Graph()
graph.add_edge("A", "B", 5)
graph.add_edge("A", "C", 10)
graph.add_edge("B", "D", 15)
graph.add_edge("C", "D", 5)
graph.add_edge("D", "E", 10)

start, goal = "A", "E"
path, cost = graph.uniform_cost_search(start, goal)
print(f"Minimum fuel cost path from {start} to {goal}: {path}, Cost: {cost}")
graph.draw_graph(path)
```

OUTPUT:-

```python
import heapq
import networkx as nx
import matplotlib.pyplot as plt

class Graph:
    def __init__(self):
        self.graph = nx.Graph()

    def add_edge(self, start, end, cost):
        self.graph.add_edge(start, end, weight=cost)

    def uniform_cost_search(self, start, goal):
        pq = [(0, start, [])]  # (cost, node, path)
        visited = {}

        while pq:
            cost, node, path = heapq.heappop(pq)

            if node in visited and visited[node] <= cost:
                continue
            visited[node] = cost
            path = path + [node]

            if node == goal:
                return path, cost  # Optimal path and its cost

            for neighbor in self.graph.neighbors(node):
                edge_cost = self.graph[node][neighbor]['weight']
                heapq.heappush(pq, (cost + edge_cost, neighbor, path))
```

```
~/workspace$ python ai.py
Minimum fuel cost path from A to E: ['A', 'C', 'D', 'E'], Cost:
25
```

GRAPH:-



5. Iterative Deepening Depth-First Search (IDDFS): Runway Allocation

Narrative: Planes landing at an airport are assigned runways incrementally based on availability and priority.

Solution Approach:

- **Algorithm:** IDDFS is useful here because it systematically searches for available runways while controlling depth.
- **State Space:** Nodes represent runway states (available or occupied), and transitions represent landing assignments.
- **Implementation Steps:**
    1. Start with a depth limit of 1 and perform Depth-First Search (DFS).
    2. If a solution is not found, increase the depth limit and restart the search.
    3. Continue until a valid runway allocation is determined.
    4. Prioritize high-priority flights (e.g., emergency landings).

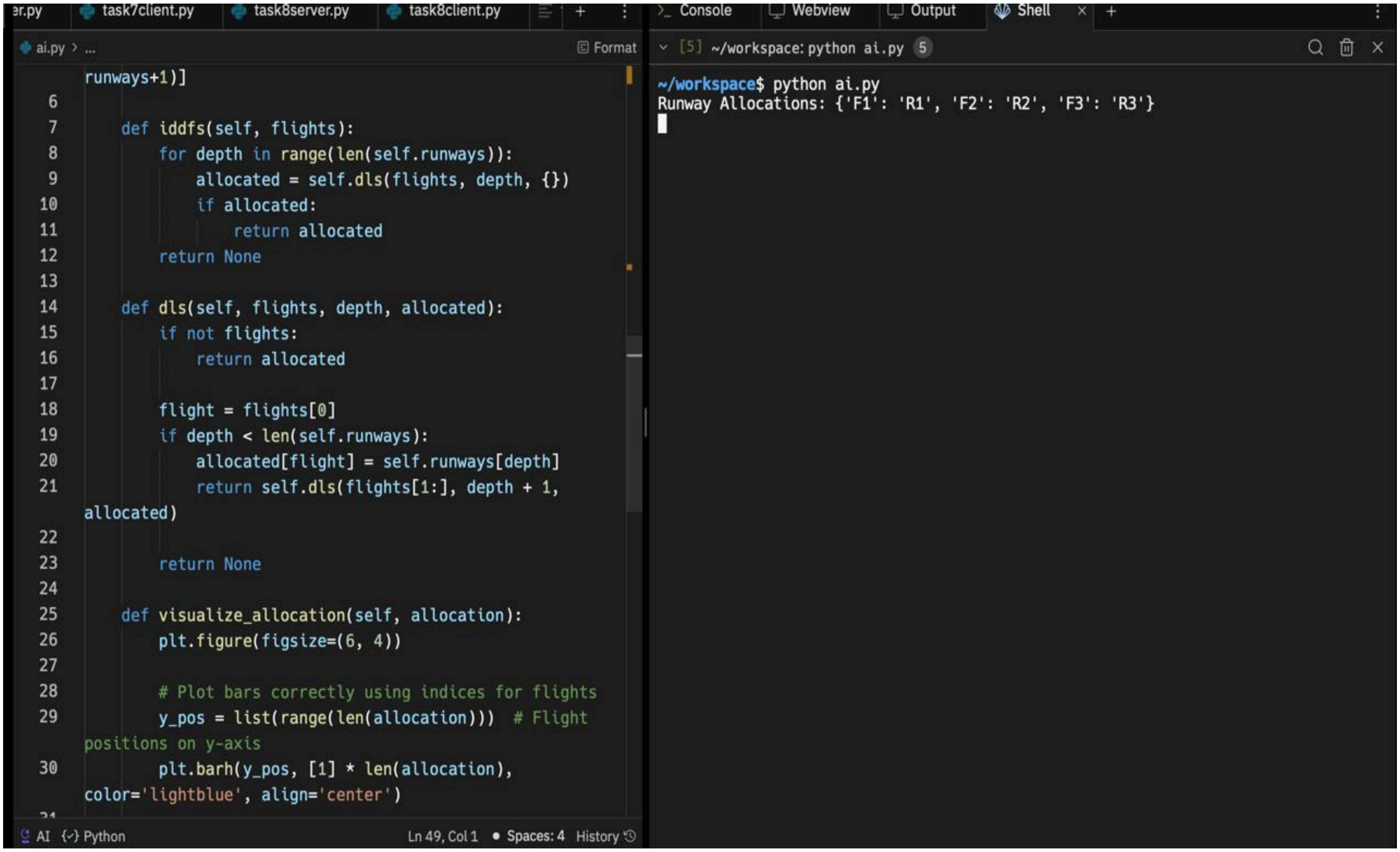
Challenges & Extensions:

- **Challenges:**
    - Avoiding assigning the same runway to multiple flights.
    - Balancing runway utilization and minimizing delays.
- **Extensions:**
    - Introducing dynamic runway closures, forcing recalculations.

PYTHON CODE:-
(ITERATIVE DEEPENING SEARCH)

```
import matplotlib.pyplot as plt

class Airport:
    def __init__(self, runways):
        self.runways = ["R" + str(i) for i in range(1, runways+1)]

    def iddfs(self, flights):
        for depth in range(len(self.runways)):
            allocated = self.dls(flights, depth, {})
            if allocated:
                return allocated
        return None

    def dls(self, flights, depth, allocated):
        if not flights:
            return allocated

        flight = flights[0]
        if depth < len(self.runways):
            allocated[flight] = self.runways[depth]
            return self.dls(flights[1:], depth + 1, allocated)

        return None

    def visualize_allocation(self, allocation):
        plt.figure(figsize=(6, 4))

        # Plot bars correctly using indices for flights
        y_pos = list(range(len(allocation)))  # Flight positions on y-axis
        plt.barh(y_pos, [1] * len(allocation), color='lightblue
```

OUTPUT:-


er.py
task7client.py
task8server.py
task8client.py
ai.py > ...
Format
runways+1)]
def iddfs(self, flights):
for depth in range(len(self.runways)):
allocated = self.dls(flights, depth, {})
if allocated:
return allocated
return None
def dls(self, flights, depth, allocated):
if not flights:
return allocated
flight = flights[0]
if depth < len(self.runways):
allocated[flight] = self.runways[depth]
return self.dls(flights[1:], depth + 1,
allocated)
return None
def visualize_allocation(self, allocation):
plt.figure(figsize=(6, 4))
# Plot bars correctly using indices for flights
y_pos = list(range(len(allocation))) # Flight
positions on y-axis
plt.barh(y_pos, [1] * len(allocation),
color='lightblue', align='center')
AI
Python
Ln 49, Col 1
Spaces: 4
History
Console
Webview
Output
Shell
[5] ~/workspace: python ai.py
~/workspace$ python ai.py
Runway Allocations: {'F1': 'R1', 'F2': 'R2', 'F3': 'R3'}

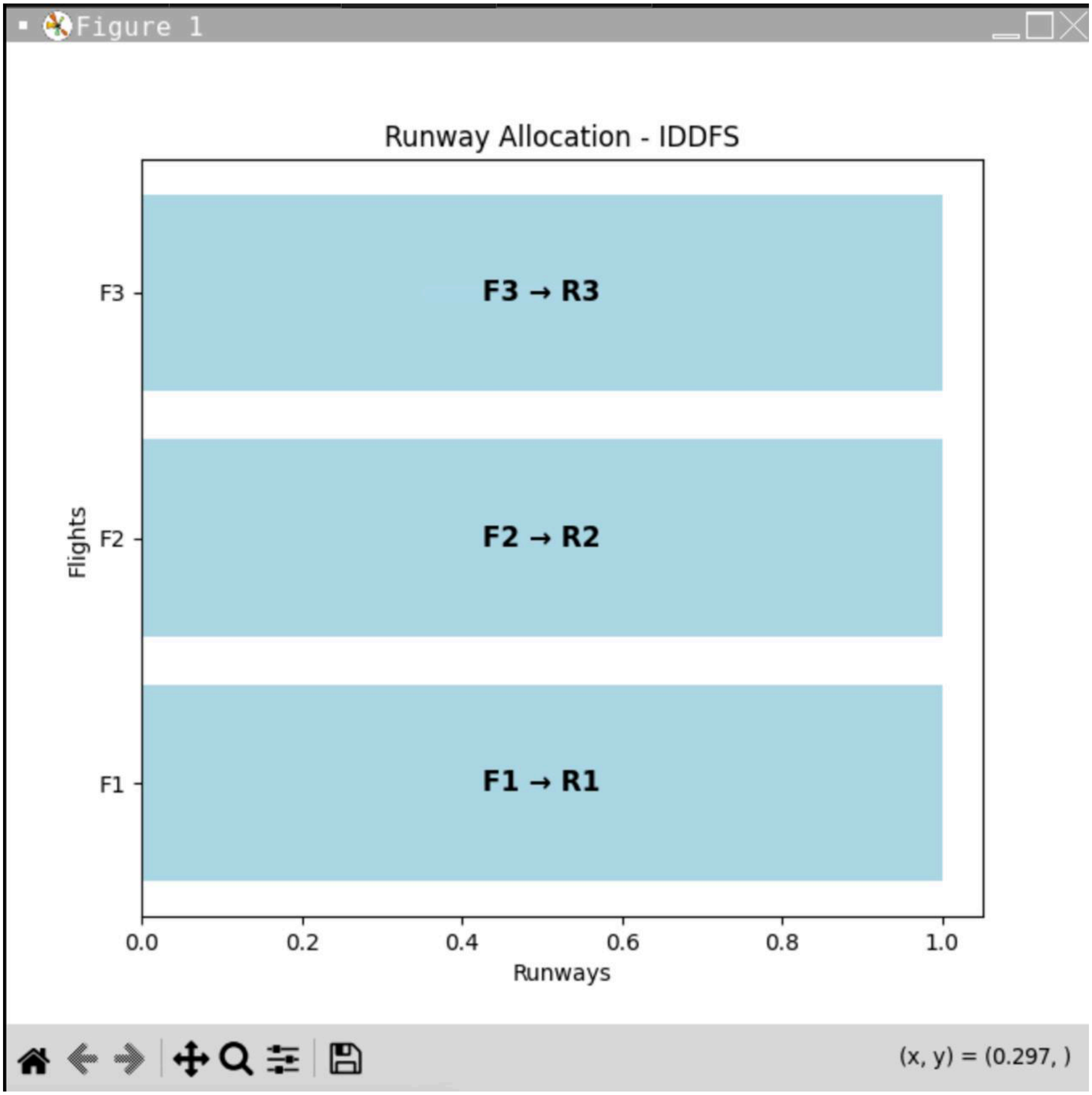


Figure 1
Runway Allocation - IDDFS
F3 → R3
F2 → R2
F1 → R1
F3
F2
F1
Flights
0.0
0.2
0.4
0.6
0.8
1.0
Runways
(x, y) = (0.297,)

6. Bidirectional Search: Inbound and Outbound Traffic Synchronization

Narrative: Coordinate inbound and outbound flights to minimize delays at a busy airport.

Solution Approach:

- **Algorithm: Bidirectional Search is used to search from both the inbound and outbound directions to meet in the middle.**
- **State Space: Nodes represent flight statuses (approaching or departing), and edges represent possible transitions.**
- **Implementation Steps:**
    1. **Start one search from inbound flights and another from outbound flights.**
    2. **Expand both searches simultaneously.**
    3. **Stop when the two searches meet, indicating an optimal synchronization.**
    4. **Resolve timing conflicts dynamically.**

Challenges & Extensions:

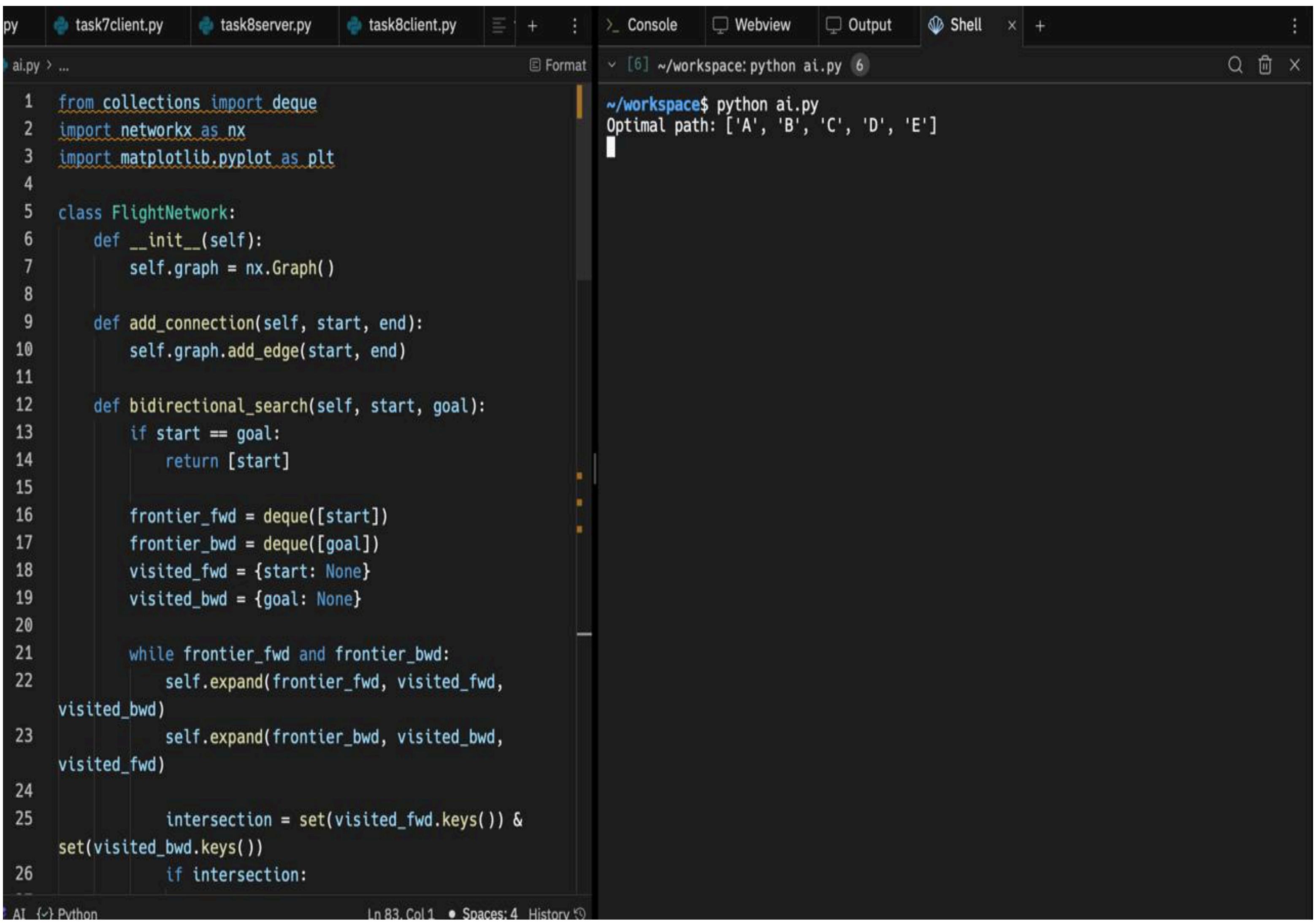
- **Challenges:**
    - **Managing timing conflicts between landing and takeoff slots.**
    - **Handling simultaneous requests from multiple flights.**
- **Extensions:**
    - **Introducing emergencies requiring prioritized landing or takeoff.**

PYTHON CODE:-
(BI-DIRECTIONAL SEARCH)

```
from collections import deque
import networkx as nx
import matplotlib.pyplot as plt

class FlightNetwork:
    def __init__(self):
        self.graph = nx.Graph()

    def add_connection(self, start, end):
        self.graph.add_edge(start, end)

    def bidirectional_search(self, start, goal):
        if start == goal:
            return [start]

        frontier_fwd = deque([start])
        frontier_bwd = deque([goal])
        visited_fwd = {start: None}
        visited_bwd = {goal: None}

        while frontier_fwd and frontier_bwd:
            self.expand(frontier_fwd, visited_fwd, visited_bwd)
            self.expand(frontier_bwd, visited_bwd, visited_fwd)

            intersection = set(visited_fwd.keys()) & set(visited_bwd.keys())
            if intersection:
                meeting_point = intersection.pop()
                return self.reconstruct_path(meeting_point, visited_fwd, visited_bwd)

        return None

    def expand(self, frontier, visited, other_visited):
        if frontier:
            node = frontier.popleft()
```

```
        for neighbor in self.graph.neighbors(node):
            if neighbor not in visited:
                visited[neighbor] = node
                frontier.append(neighbor)
            if neighbor in other_visited:
                return

    def reconstruct_path(self, meeting_point, visited_fwd, visited_bwd):
        path_fwd, path_bwd = [], []

        # Construct forward path
        node = meeting_point
        while node is not None:
            path_fwd.append(node)
            node = visited_fwd[node]

        # Construct backward path
        node = visited_bwd[meeting_point]
        while node is not None:
            path_bwd.append(node)
            node = visited_bwd[node]

        return path_fwd[::-1] + path_bwd  # Reverse fwd path and append backward
path

    def draw_graph(self, path=[]):
        pos = nx.spring_layout(self.graph)

        plt.figure(figsize=(6, 6))
        nx.draw(self.graph, pos, with_labels=True, node_color='lightblue',
edge_color='gray', node_size=2000, font_size=10)

        if path:
            edges = list(zip(path, path[1:]))
            nx.draw_networkx_edges(self.graph, pos, edgelist=edges,
edge_color='red', width=2, alpha=0.8)
```

```python
        plt.title("Bidirectional Search - Optimal Path")
        plt.show()

# Example usage
network = FlightNetwork()
network.add_connection("A", "B")
network.add_connection("B", "C")
network.add_connection("C", "D")
network.add_connection("D", "E")

start, goal = "A", "E"
path = network.bidirectional_search(start, goal)
print("Optimal path:", path)
network.draw_graph(path)
```



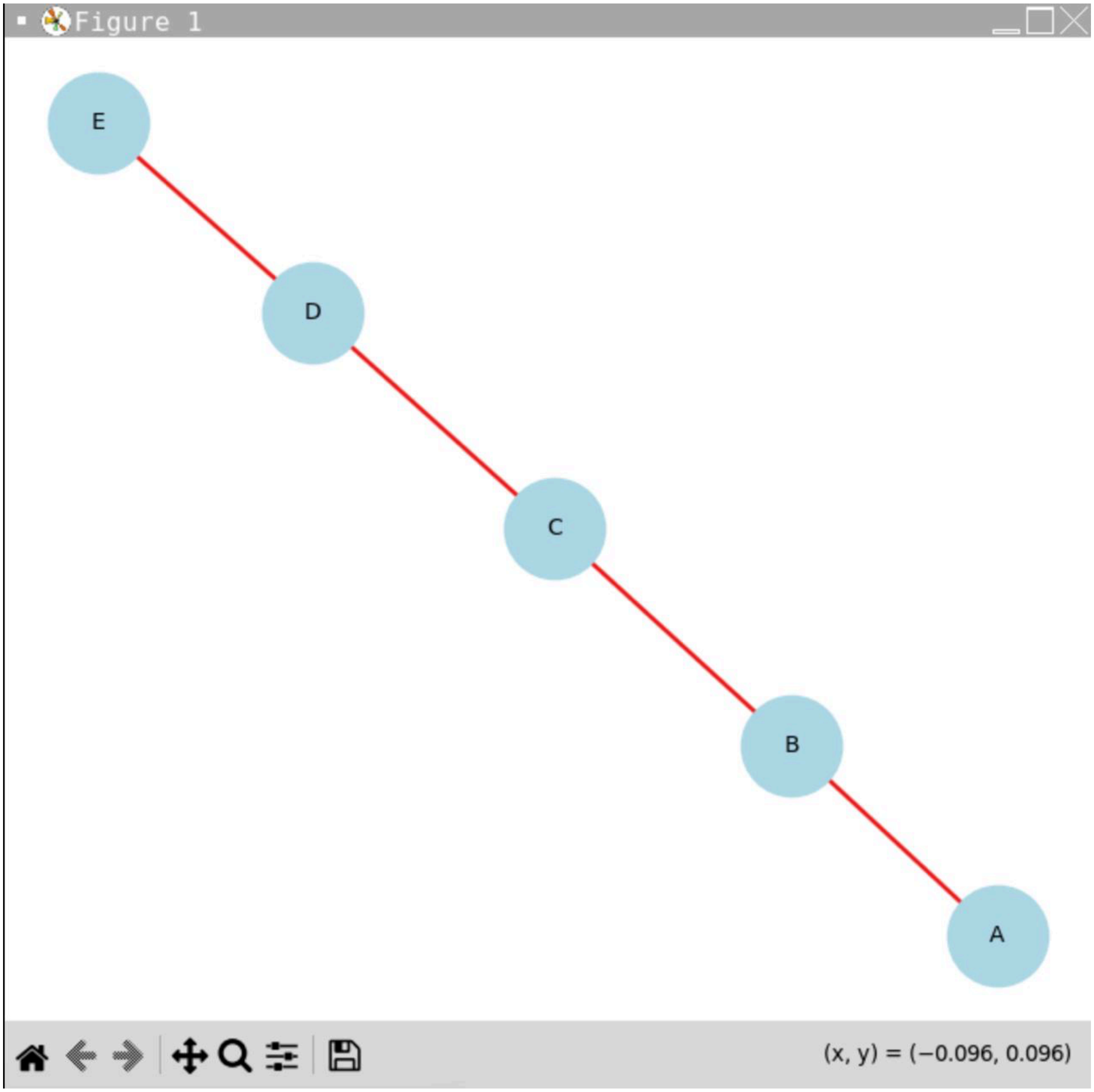


Figure 1
E
D
C
B
A
(x, y) = (−0.096, 0.096)

🔹 Real-World Applications

✈ Air Traffic Control (ATC) Towers use simulations to predict and avoid congestion.
✈ Autonomous Drones & UAVs use AI-based air traffic management systems.
✈ Flight Planning Systems use algorithms to optimize airline schedules.

🔹 Conclusion

Airplane Traffic Simulation Management is a critical field that enhances safety, efficiency, and reliability in aviation. Using AI, search algorithms, and optimization techniques, it ensures a seamless experience for passengers, airlines, and air traffic controllers.