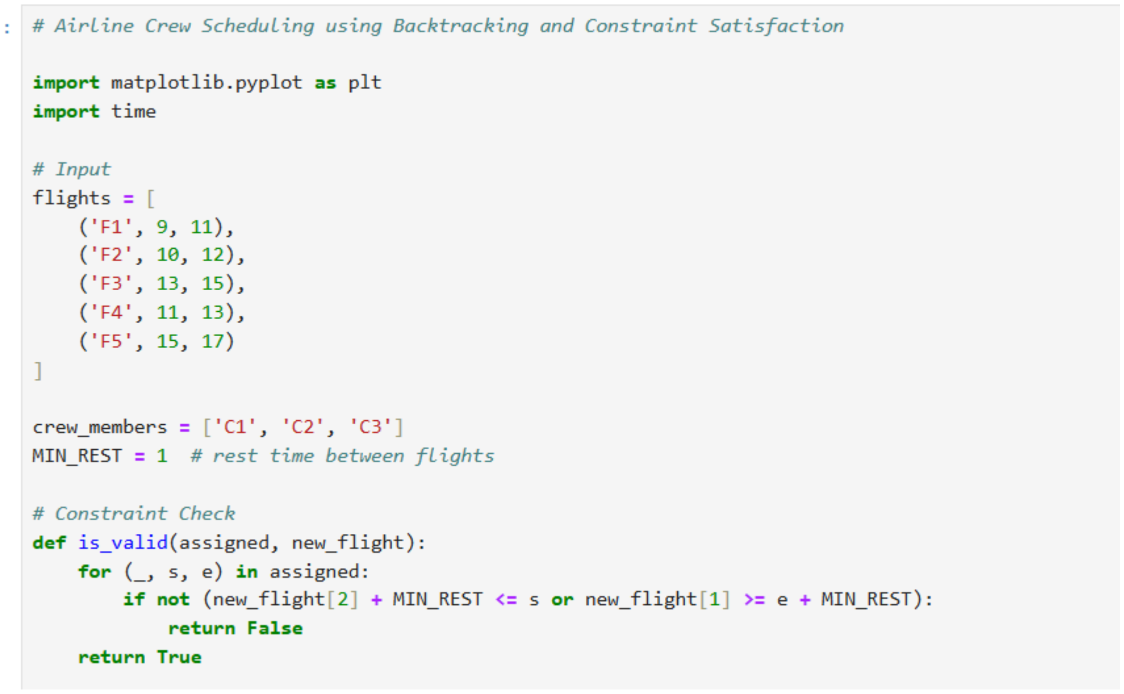
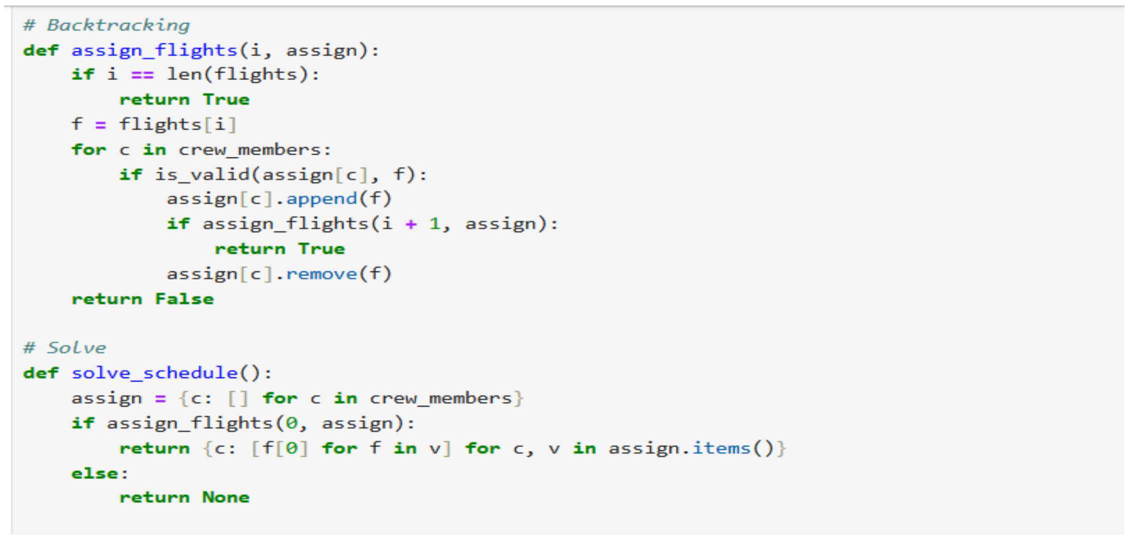
Problem: Airline Crew Scheduling Using Backtracking and Constraint Satisfaction

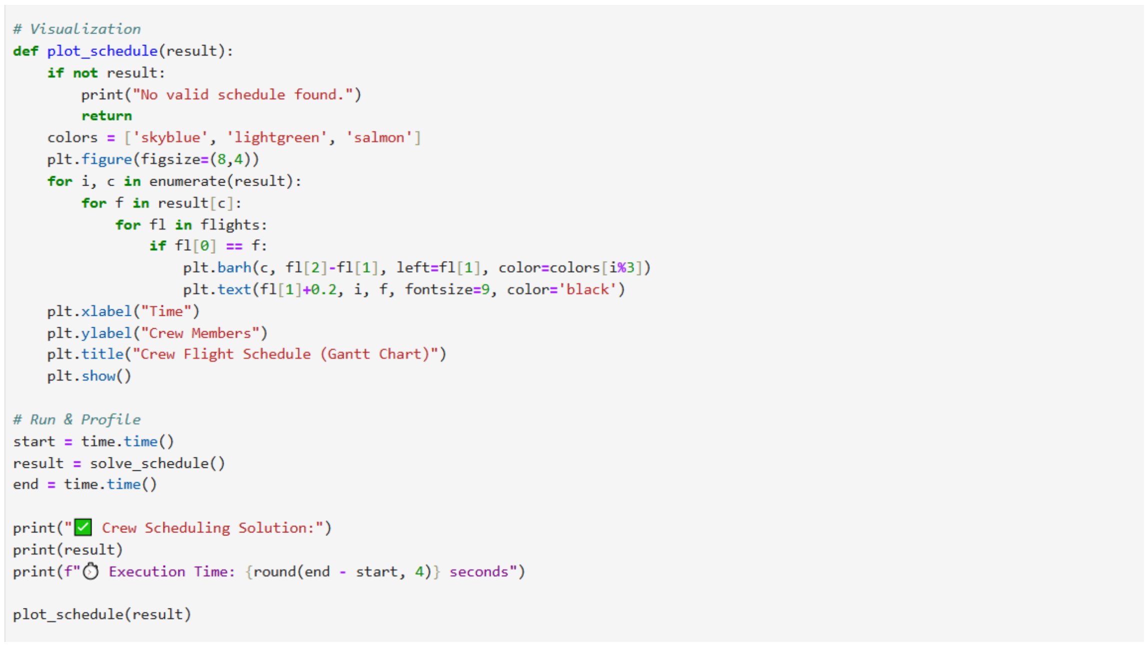
Source Code:



```python
# Airline Crew Scheduling using Backtracking and Constraint Satisfaction

import matplotlib.pyplot as plt
import time

# Input
flights = [
    ('F1', 9, 11),
    ('F2', 10, 12),
    ('F3', 13, 15),
    ('F4', 11, 13),
    ('F5', 15, 17)
]

crew_members = ['C1', 'C2', 'C3']
MIN_REST = 1  # rest time between flights

# Constraint Check
def is_valid(assigned, new_flight):
    for (_, s, e) in assigned:
        if not (new_flight[2] + MIN_REST <= s or new_flight[1] >= e + MIN_REST):
            return False
    return True
```



```python
# Backtracking
def assign_flights(i, assign):
    if i == len(flights):
        return True
    f = flights[i]
    for c in crew_members:
        if is_valid(assign[c], f):
            assign[c].append(f)
            if assign_flights(i + 1, assign):
                return True
            assign[c].remove(f)
    return False

# Solve
def solve_schedule():
    assign = {c: [] for c in crew_members}
    if assign_flights(0, assign):
        return {c: [f[0] for f in v] for c, v in assign.items()}
    else:
        return None
```



```python
# Visualization
def plot_schedule(result):
    if not result:
        print("No valid schedule found.")
        return
    colors = ['skyblue', 'lightgreen', 'salmon']
    plt.figure(figsize=(8,4))
    for i, c in enumerate(result):
        for f in result[c]:
            for fl in flights:
                if fl[0] == f:
                    plt.barh(c, fl[2]-fl[1], left=fl[1], color=colors[i%3])
                    plt.text(fl[1]+0.2, i, f, fontsize=9, color='black')
    plt.xlabel("Time")
    plt.ylabel("Crew Members")
    plt.title("Crew Flight Schedule (Gantt Chart)")
    plt.show()

# Run & Profile
start = time.time()
result = solve_schedule()
end = time.time()

print("✅ Crew Scheduling Solution:")
print(result)
print(f"⏱ Execution Time: {round(end - start, 4)} seconds")

plot_schedule(result)
```

1. Input

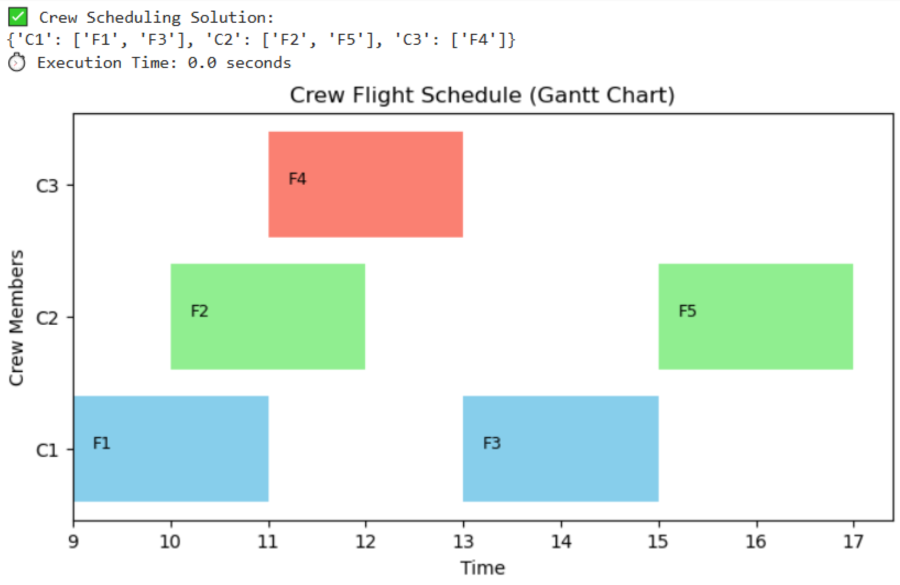
- **Flights List:**
  Each flight is represented as a tuple (Flight ID, Start Time, End Time)
- **Crew Members:**
- Each crew member can be assigned multiple flights provided that flight timings do not overlap, and rest time constraints are met.

2. Approach

- A **constraint checker function** verifies that:
  - No two flights assigned to the same crew overlap in time.
  - A **minimum rest time (e.g., 1 hour)** is maintained between two consecutive flights.

- The algorithm uses **recursive backtracking**:
    - Assign flights sequentially to available crew members.
    - If a conflict occurs, the algorithm backtracks and tries another possible assignment.
    - Continue until all flights are successfully assigned or no valid combination exists.
- A **cost parameter** can be added to minimize the total assignment cost.

3. Output



```
✅ Crew Scheduling Solution:
{'C1': ['F1', 'F3'], 'C2': ['F2', 'F5'], 'C3': ['F4']}
⏱ Execution Time: 0.0 seconds
```

4. Analysis

- **Nature of Problem:** Crew scheduling is an **NP-hard combinatorial optimization problem** — the number of possible flight assignments increases exponentially with input size.
- **Why Backtracking Works Only for Small n:** Backtracking checks all valid combinations; as flight count increases, the search tree grows rapidly, making it infeasible for large datasets.
- **Time Complexity:** *O(k×2n)O(k \times 2^n)*O(k×2n), where *n* = number of flights, *k* = number of crew members.
- **Possible Improvements:**
    - Use **Heuristics or Greedy algorithms** for faster approximate solutions.
    - Apply **Integer Linear Programming (ILP)** or **Constraint Programming** for optimization.
    - Utilize **Google OR-Tools** or **CP-SAT solvers** for real-world scale scheduling.

5. Visualization

- A **Gantt Chart** can be plotted using `matplotlib` to visualize each crew member's flight timeline.
- The chart helps identify overlapping flights and gaps (rest periods).
- Another plot can be created showing **execution time vs. number of flights** to demonstrate how quickly complexity grows with input size.