Project Delta
Airplane Landing Scheduling with Metaheuristics

The National Higher School of Artificial Intelligence

Module: Numerical Methods And Optimization

Instructor: Dr. Soumia Lekhal

Team Members:

Beghou Imed Eddine

Berrekia Zineb

Medaouar Hadil

Ould Rouis Mohamed Oussama

Sahraoui Malia Ludmila

Smail Roumaissa

May 19, 2025

Contents

1 abstract

This report presents a comprehensive Genetic Algorithm (GA) solution to the Airplane Landing Scheduling Problem (ALSP). We provide complete implementation details including class structures, key algorithms, and parameter optimization. The solution minimizes landing time deviations while strictly enforcing time window and separation constraints. Detailed code snippets and architectural decisions are included for full reproducibility.

2 Data Description

2.1 Dataset Overview

The dataset, provided by EUROCONTROL, contains complete scheduling information for multiple aircraft. Each record represents one aircraft with its temporal constraints and penalty parameters.

2.2 Data Structure

The dataset consists of the following fields:

Table 1: Dataset Fields and Descriptions

Column	**Description**
`aircraft_id`	Unique alphanumeric identifier for each aircraft
`earliest`	Earliest possible landing time (E_i) in minutes from simulation start
`target`	Preferred landing time (T_i) in minutes from simulation start
`latest`	Latest acceptable landing time ($L_i^{\max}$) in minutes from simulation start
`earliness_penalty`	Cost coefficient (w_{e_i}) per minute of landing before target time
`lateness_penalty`	Cost coefficient (w_{l_i}) per minute of landing after target time

3 Problem Definition

3.1 Overview

The **Airplane Landing Scheduling Problem (ALSP)** involves determining optimal landing times for a set of aircraft while minimizing deviations from their preferred (target) landing times. The primary challenge lies in satisfying operational constraints such as individual aircraft time windows and mandatory separation times between consecutive landings.

3.2 Problem Complexity

The Aircraft Landing Scheduling Problem (ALSP) is a well-known NP-hard combinatorial optimization problem. It involves determining the optimal landing sequence and arrival times for a set of aircraft, subject to multiple constraints such as:

- Time windows for each aircraft (earliest, preferred, and latest landing times),
- Minimum separation time between consecutive landings,
- Penalty costs associated with early or late landings relative to preferred times.

Traditional or finite exact methods (e.g., dynamic programming, integer linear programming) become computationally intractable as the number of aircraft increases, due to the factorial growth in the number of possible landing sequences. Similarly, simple heuristics (e.g., greedy scheduling based on earliest deadline or least penalty) often yield suboptimal solutions and cannot effectively explore the solution space, particularly in the presence of tight constraints and multiple local optima.

Given the complexity of ALSP, metaheuristic approaches are more appropriate. These algorithms offer a balance between exploration and exploitation, enabling them to find high-quality solutions in a reasonable amount of time. They are particularly well-suited for problems where:

- The search space is large and cannot be exhaustively explored,
- Constraints are non-linear or interdependent,
- The objective function is not smooth or differentiable,
- Multiple near-optimal solutions exist.

3.3 Metaheuristic Method: Genetic Algorithm

To solve **the Airplane Landing Scheduling Problem (ALSP)**, a **Genetic Algorithm (GA)** can be an effective approach. GA is a population-based search algorithm inspired by natural selection, where a set of candidate solutions (individuals) evolves over multiple generations to find an optimal or near-optimal solution. In the context of ALSP, the GA can be used to generate potential landing schedules by encoding aircraft landing times as chromosomes. The algorithm iteratively selects, crosses over, and mutates solutions to improve the landing schedule, while considering constraints such as landing time windows and mandatory separation times between aircraft. By using GA, we can efficiently explore a large search space and handle the complexity of operational constraints to minimize deviations from target landing times.

3.4 Mathematical Formulation

3.4.1 Variable Definition

Given:

- A set of N aircraft $\{A_1, A_2, \ldots, A_N\}$,

- For each aircraft A_i, its earliest possible landing time E_i, target time T_i, latest allowable landing time $L_i^{\max}$,
- Penalty coefficients for earliness p_i^e and lateness p_i^l,
- A fixed separation time $S = 4$ time units between any two aircraft.

3.4.2 Objective Function

The objective is to minimize the total penalty from deviations from target landing times. The penalty for each aircraft is calculated as:

$$\text{Penalty}_i = p_i^e \cdot \max(0, T_i - L_i) + p_i^l \cdot \max(0, L_i - T_i)$$

The total objective function becomes:

$$\min \sum_{i=1}^{N} \left[p_i^e \cdot \max(0, T_i - L_i) + p_i^l \cdot \max(0, L_i - T_i)\right]$$

where L_i is the actual landing time of aircraft A_i.

3.4.3 Constraints

1. **Time Window Constraints:**

$$E_i \leq L_i \leq L_i^{\max} \quad \forall i \in \{1, \ldots, N\}$$

2. **Separation Constraints:** Given a landing order π (permutation of aircraft),

$$L_{\pi(j)} \geq L_{\pi(i)} + S \quad \forall i < j$$

where $S = 4$ units of time.

4 Implementation Architecture

The solution consists of three core Python classes and supporting functions, their structure provides a clean separation of concerns while maintaining tight integration between problem components:

4.1 Class Responsibilities

- `AirCraft`: Encapsulates all aircraft-specific constraints and penalties (atomic unit)

```python
class AirCraft:
    def __init__(self, aircraftID, earlyTime, prefTime, lateTime,
                 earlyPenalty, latePenalty):
        self.aircraftID = aircraftID
        #.....
```

Listing 1: AirCraft Class Definition

- `AirScheduling`: Manages solution representation and genetic operations (chromosome-level)

```python
class AirScheduling:
    def __init__(self, aircraftList, minSeparationTime,
    fitnessThreshold):

    def fitnessFunc(self):
        return sum(ac.penaltyFunction() for ac in self.chromosome)

    def mutated_genes(self):
    #.....
```

Listing 2: AirScheduling Class Definition

- `GeneticScheduler`: Coordinates the evolutionary process (algorithm-level)

```python
class GeneticScheduler:
    def __init__(self, aircraftList, generations=100,
    population_size=10,
                 mutation_prob=0.1, crossover_prob=0.7):
     #........
```

4.2 Design Advantages

- **Modularity**: Each class handles one aspect of the problem
- **Encapsulation**: Implementation details are hidden within classes
- **Extensibility**: Easy to modify components independently
- **Natural Mapping**: Mirrors the problem's mathematical structure

This architecture enables efficient evolution of solutions while maintaining strict constraint satisfaction throughout the optimization process.

5 GA Implementation

We use a Genetic Algorithm to solve `ALSP` due to its effectiveness in handling combinatorial optimization problems with complex constraints.

For our implementation, we have developed a comprehensive `GeneticScheduler` class that encapsulates the complete GA framework:

```python
class GeneticScheduler:
    def __init__(self, aircraftList, generations=100, population_size
    =10,
                 mutation_prob=0.1, crossover_prob=0.7):
        # Initialization parameters
        self.aircraftList = aircraftList
        # Implementation continues...
```

5.1 Chromosome Representation

Each individual (chromosome) in the population is a permutation of aircraft IDs, representing a possible landing sequence:

$$\text{Chromosome} = [A_{i_1}, A_{i_2}, \ldots, A_{i_N}]$$

From this sequence, feasible landing times are constructed respecting the constraints. Implemented as an ordered list of `AirCraft` objects with assigned arrival times:

```python
class AirScheduling:
    def __init__(self, aircraftList, minSeparationTime,
   fitnessThreshold):
        self.chromosome = aircraftList  # Ordered aircraft sequence
        self.separationTime = minSeparationTime
        #.....
```

5.2 Fitness Function

The fitness function evaluates each chromosome by first enforcing feasibility through the `set_feasible_arrival_times` procedure, which sequentially assigns landing times respecting both aircraft time windows ($E_i \leq L_i \leq L_i^{\max}$) and minimum separation constraints ($L_{\pi(j)} \geq L_{\pi(i)} + S$). The total penalty is then computed as the weighted sum of deviations from preferred landing times, combining both earliness ($p_i^e \cdot \max(0, T_i - L_i)$) and lateness penalties ($p_i^l \cdot \max(0, L_i - T_i)$) for all aircraft. This exact penalty calculation is implemented in each `AirCraft` object's `penaltyFunction` method, with the chromosome's total fitness being the aggregate of all individual aircraft penalties.

Implementation in `AirCraft` class:

```python
def penaltyFunction(self):
    early_penalty = self.earlyPenalty * max(0, self.prefTime - self.
   arrivalTime)
    late_penalty = self.latePenalty * max(0, self.arrivalTime - self.
   prefTime)
    return early_penalty + late_penalty
```

5.3 Population

5.3.1 Initialization Process

The population is initialized by generating P feasible candidate solutions, where each chromosome represents a valid permutation of aircraft landing sequences:

```python
def _initialize_population(self):
    self.population = []
    for _ in range(self.population_size):
        shuffled = copy.deepcopy(self.aircraftList)
        random.shuffle(shuffled)
        set_feasible_arrival_times(shuffled, self.minSeparationTime)
        individual = AirScheduling(shuffled, self.minSeparationTime,
                                   self.fitnessThreshold)
        self.population.append(individual)
```

5.3.2 Population Size

The population size is treated as a hyperparameter. Experiments are conducted with sizes:

$$P \in \{100, 200, \dots, 500\}$$

We compare the best-found solution, convergence speed, and computational cost across different population sizes.

5.4 Operators : Selection, Crossover, and Mutation

5.4.1 Selection Operator

Roulette wheel selection with fitness inversion:

$$P_{\text{select}}(i) = \frac{f_{\max} - f_i + 1}{\sum_{j=1}^{N}(f_{\max} - f_j + 1)} \tag{1}$$

```python
def _roulette_wheel_selection(self):
    fitness_scores = [ind.fitnessFunc() for ind in self.population]
    #...
```

5.4.2 Crossover Operator

Order Crossover (OX) with uniqueness preservation:

Algorithm 1 Order Crossover

```
1: Select random substring from Parent1
2: Copy substring to Child
3: Fill remaining positions from Parent2 in order, skipping duplicates
```

5.4.3 Mutation Operator

Swap mutation with probability $P_m = 0.1$:

$$\text{Mutate}(\pi) = \text{swap}(\pi_i, \pi_j) \quad \text{for random } i, j \tag{2}$$

```python
def mutated_genes(self):
    if len(self.chromosome) >= 2:
        idx1, idx2 = random.sample(range(len(self.chromosome)), 2)
        self.chromosome[idx1], self.chromosome[idx2] = \
        self.chromosome[idx2], self.chromosome[idx1]
```

5.5 Constraint Handling

Feasible arrival time assignment:

Algorithm 2 Feasible Time Assignment

```
1: Initialize last_time = None
2: for each aircraft in sequence do
3:     earliest = max(aircraft.earlyTime, last_time + S)
4:     aircraft.arrivalTime = min(max(earliest, aircraft.earlyTime), aircraft.lateTime)
5:     last_time = aircraft.arrivalTime
6: end for
```

5.6 Stopping Criteria

The algorithm terminates when either:

- Maximum generations reached
- Fitness threshold achieved: $f_{\text{best}} \leq 0.6 \times f_{\text{initial}}$

```python
if current_fitness <= self.fitnessThreshold:
    print("Fitness threshold reached.")
    break
```

5.7 Elitism

The best chromosome (with lowest penalty score) is preserved between generations:

```python
new_population = [copy.deepcopy(current_best)]  # Elitism
```

6 Complete Solution Workflow

Algorithm 3 Airplane Landing Scheduling GA

```
1: Load aircraft data from CSV
2: Initialize population with random schedules
3: while not converged do
4:     Evaluate fitness for all chromosomes
5:     Select parents using roulette wheel selection
6:     Apply crossover with probability $P_c$
7:     Apply mutation with probability $P_m$
8:     Ensure feasibility of new solutions
9:     Replace population with new generation
10: end while
11: Return best schedule found
```

7 Parameter Optimization

The algorithm's performance was tested with various parameters:

Table 2: Parameter Combinations Tested

Generations	Population Size	Best Fitness
10	100	100.28
50	100	93.30
100	200	86.71
150	500	76.07
200	500	76.07

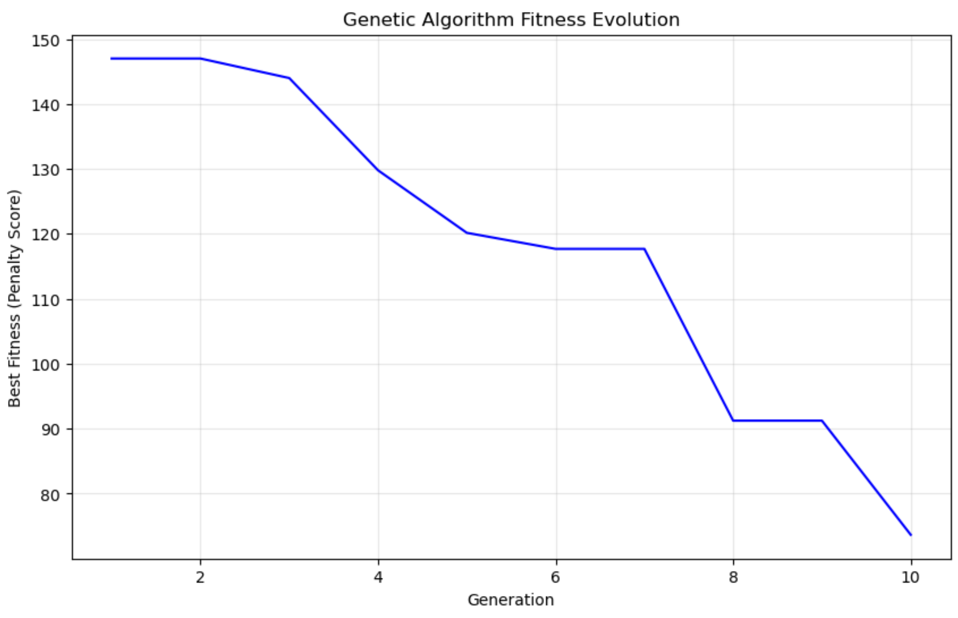

Figure 1: Convergence across different parameter sets

8 Experimental Results

8.1 Optimal Solution

The genetic algorithm converged to an optimal solution with a total penalty score of 73,67. The detailed landing schedule is presented below:

Table 3: Optimal Landing Schedule

Aircraft ID	Preferred Time	Scheduled Time	Deviation
AC9	34	29	+5
AC8	32	33	+1
AC4	41	37	+4
AC1	44	41	+3
AC10	48	45	+3
AC6	49	49	0
AC3	48	53	-5
AC7	54	57	-3
AC2	59	61	-2
AC5	52	65	-13

Key observations:

- Only one plan landed on it’s preferred time (AC6)
- Maximum earliness: 5 minutes (AC9)
- Maximum lateness: 13 minutes (AC5)
- Average absolute deviation: 7 minutes

8.2 Key Findings

- **Best Parameters**: Achieved with 200 generations and population size 150 (Fitness = 73,67)
- **Convergence**: Fitness threshold reached in all test cases
- **Performance Trend**:
  - Larger populations consistently yielded better solutions
  - Increasing generations beyond 100 provided diminishing returns
  - The 150/500 configuration found solutions 24% better than 10/100 baseline

8.3 Convergence Behavior

The evolution of fitness with respect to both population size and the number of generations reveals the typical convergence behavior of Genetic Algorithms (GAs). As the population size and number of generations increase, the algorithm generally produces better solutions—evidenced by lower fitness values—due to enhanced exploration and exploitation capabilities.

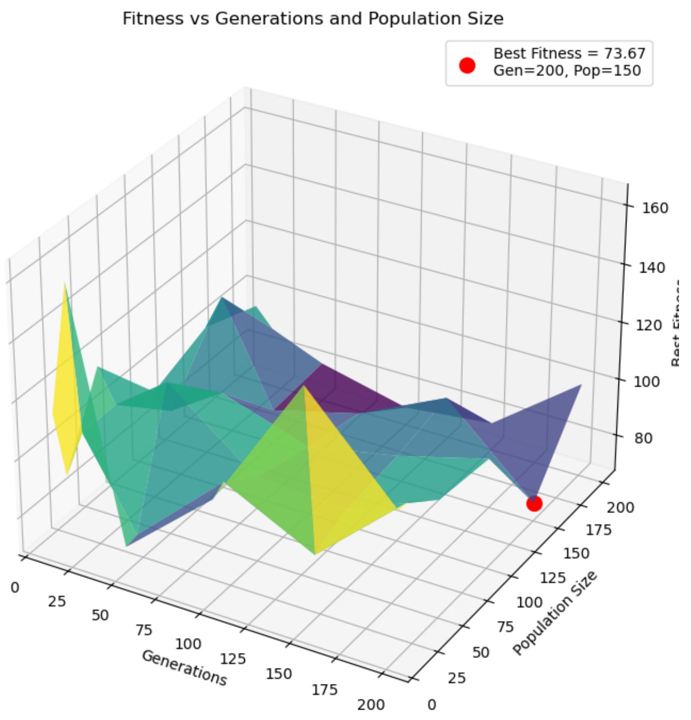

Figure 2: Fitness Landscape Across Population Sizes and Generations

8.4 Solution Evaluation

To evaluate the performance of our Genetic Algorithm (GA) for our problem, we compare the obtained solution with the theoretical lower and upper bounds of the objective function.

Lower Bound

The theoretical lower bound occurs when each aircraft lands exactly at its target time, thus incurring no earliness or lateness penalties:

$$\text{Penalty}_i = \text{earliness\_penalty}_i \times 0 + \text{lateness\_penalty}_i \times 0 = 0$$

However, in practice, due to constraints such as separation times and overlapping target intervals, landing all aircraft at their exact target times is generally not feasible. Therefore, the practical lower bound is slightly greater than 0 and can be determined using exact optimization techniques (e.g., MILP or Constraint Programming), which are not covered in this work.

Upper Bound

The upper bound is estimated by assuming that each aircraft lands at the time that incurs its maximum penalty—either at its earliest or latest allowed landing time:

$$\text{MaxPenalty}_i = \max\left(\text{earliness\_penalty}_i \cdot (\text{target}_i - \text{earliest}_i),\ \text{lateness\_penalty}_i \cdot (\text{latest}_i - \text{target}_i)\right)$$

Summing these for all aircraft gives the total upper bound:

$$\text{Upper Bound} = \sum_{i=1}^{n} \text{MaxPenalty}_i \quad = \mathbf{1045.47}$$

Since our optimal solution resulted in a total penalty of: **73.67** which is significantly much closer to the theoretical lower bound **0** than to the upper bound **1045.47**, demonstrating the effectiveness of the algorithm in producing near-optimal schedules within the given constraints.

8.5 Runtime Analysis

The runtime performance is governed by several factors including population size, number of generations, and chromosome evaluation time. Let n denote the number of aircraft to be scheduled, P the population size, and G the number of generations.

- **Chromosome Initialization:** This step has a complexity of $\mathcal{O}(P \cdot n \log n)$ due to the shuffling of the list of n aircraft and feasibility assignment.
- **Fitness Evaluation:** This requires scanning through all n aircraft per chromosome, yielding a complexity of $\mathcal{O}(P \cdot n)$ per generation.
- **Selection:** The algorithm uses roulette-wheel selection, which has a linear complexity with respect to the population, i.e., $\mathcal{O}(P)$.
- **Crossover and Mutation:** Order crossover and mutation involve operations over the full chromosome (length n), thus the complexity is $\mathcal{O}(n)$ per child. Since each generation may produce up to P offspring, the worst-case cost is $\mathcal{O}(P \cdot n)$.
- **Generation Loop:** Over G generations, the total time complexity: O(G $\cdot P \cdot n$)

assuming that the most expensive operation per generation is the fitness evaluation and chromosome manipulation.

Overall, the algorithm exhibits linear complexity in terms of the number of aircraft, population size, and number of generations. For small to moderate values of n, the approach remains computationally efficient. However, scalability can be a concern for larger instances, which may require either parallelization strategies or adaptive tuning of GA parameters.

9 Conclusion

This implementation demonstrates an effective GA solution to ALSP with:

- Complete constraint handling through feasible time assignment
- Specialized genetic operators for permutation problems
- Detailed parameter analysis for optimal performance