Comparative Metaheuristic Strategies for Airline Crew Scheduling

A Study of Simulated Annealing and Genetic Algorithm Approaches

Evolutionary Computation

Author: Max Hart

mah422@student.bham.ac.uk

Professors: Dr Shan He & Dr Per Kristian Lehre

March 3, 2025

Contents

1 Introduction

The airline crew scheduling problem is a complex optimisation challenge, in which it is often framed as a Set Partitioning Problem (SPP). In this setup, we must have each flight leg assigned to exactly one valid crew rotation. This must be done while satisfying various operational constraints. Due to the rapid growth in the number of possible assignments in comparison to the problem size, exact methods can struggle to scale. This has led to the exploration of metaheuristic algorithms that offer practical, feasible (and often near-optimal) solutions within reasonable time. These algorithms are particularly well-suited to the SPP, as they can efficiently navigate the vast solution space and balance the trade-offs between cost and constraint satisfaction.

In this project, we implement three such algorithms from scratch:

- **Simulated Annealing (SA)**
- **Standard Binary Genetic Algorithm (BGA)**
- **Improved Binary Genetic Algorithm (BGA)**

We will test each of these algorithms for 30 runs on three instances from the OR-Library (sppnw41, sppnw42, sppnw43) and compare their performance in terms of feasibility, solution cost and overall robustness. Each algorithm progressively improves in sophistication, with the Improved BGA incorporating several problem-specific enhancements to boost efficiency and constraint satisfaction. Specifically, we draw on ideas from Chu and Beasley [1] and Runarsson and Yao [2], including pseudo-random initialization, heuristic improvement operators, and stochastic ranking for constraint handling.

Our main objective throughout this report is to investigate how these metaheuristic techniques will manage the complex challenges of the SPP, and to provide a comprehensive comparison of their performance. Through this, we aim to gain insights into the strengths and weaknesses of each algorithm, and to identify the most effective strategies for solving airline crew scheduling problems.

2 Metaheuristic Algorithms for the SPP

In this section, we will provide a comprehensive overview of the three metaheuristic algorithms developed for solving the SPP. Each algorithm has been implemented from scratch in Python and is designed to find high-quality solutions to the airline crew scheduling problem. For each algorithm, we will provide a brief introduction outlining its underlying principles and technical merits, followed by the corresponding pseudocode. Detailed flowcharts will be presented on subsequent pages to visually encapsulate the step-by-step processes. This thorough explanation is intended to clarify both the operational mechanics and the innovative aspects of our methods.

2.1 Simulated Annealing

Simulated Annealing is a local search technique that iteratively improves a candidate solution by exploring its neighbourhood. It uses a temperature parameter to probabilistically accept inferior solutions, thereby enabling the algorithm to escape local optima. This simple yet effective approach balances exploration and exploitation through gradual cooling.

2.1.1 Pseudocode for Simulated Annealing

The pseudocode below outlines the Simulated Annealing algorithm for the SPP, detailing random solution initialization, neighbour generation via bit-flipping, acceptance via direct improvement or the Metropolis criterion, and temperature cooling.

Algorithm 1 SimulatedAnnealing(T, α, maxIter, penaltyFactor)

```
1: **Initialisation:**
2: $x \leftarrow$ RandomSolution()                          ▷ Generate a random binary solution
3: $F(x) \leftarrow$ PenaltyFitness($x$, penaltyFactor)      ▷ Cost + coverage violations * penalty
4: $x_{\text{best}} \leftarrow x$;   $F_{\text{best}} \leftarrow F(x)$
5: **for** $iter = 1$ to maxIter **do**
6:     **Neighbour generation:**
7:     $x' \leftarrow$ FlipOneRandomBit($x$)                ▷ Create neighbour by flipping exactly one bit
8:     **Evaluate neighbour:**
9:     $F(x') \leftarrow$ PenaltyFitness($x'$, penaltyFactor)
10:    **if** $F(x') < F(x)$ **then**
11:        $x \leftarrow x'$                                 ▷ If neighbour is better, accept it outright
12:    **else**
13:        $\Delta \leftarrow F(x') - F(x)$                  ▷ Compute the increase in fitness
14:        **Metropolis criterion:**
15:        **if** rand() $< e^{-\Delta/T}$ **then**
16:            $x \leftarrow x'$                             ▷ Accept worse solution with probability $e^{-\Delta/T}$
17:    **Update global best:**
18:    **if** $F(x) < F_{\text{best}}$ **then**
19:        $x_{\text{best}} \leftarrow x$;   $F_{\text{best}} \leftarrow F(x)$
20:    **Cool down:**
21:    $T \leftarrow \alpha \times T$
22: **return** $(x_{\text{best}}, F_{\text{best}})$
```

2.1.2 Flowchart for Simulated Annealing

The flowchart visually represents the step-by-step process of the Simulated Annealing algorithm, from initialisation and neighbour evaluation to acceptance decisions and temperature cooling.

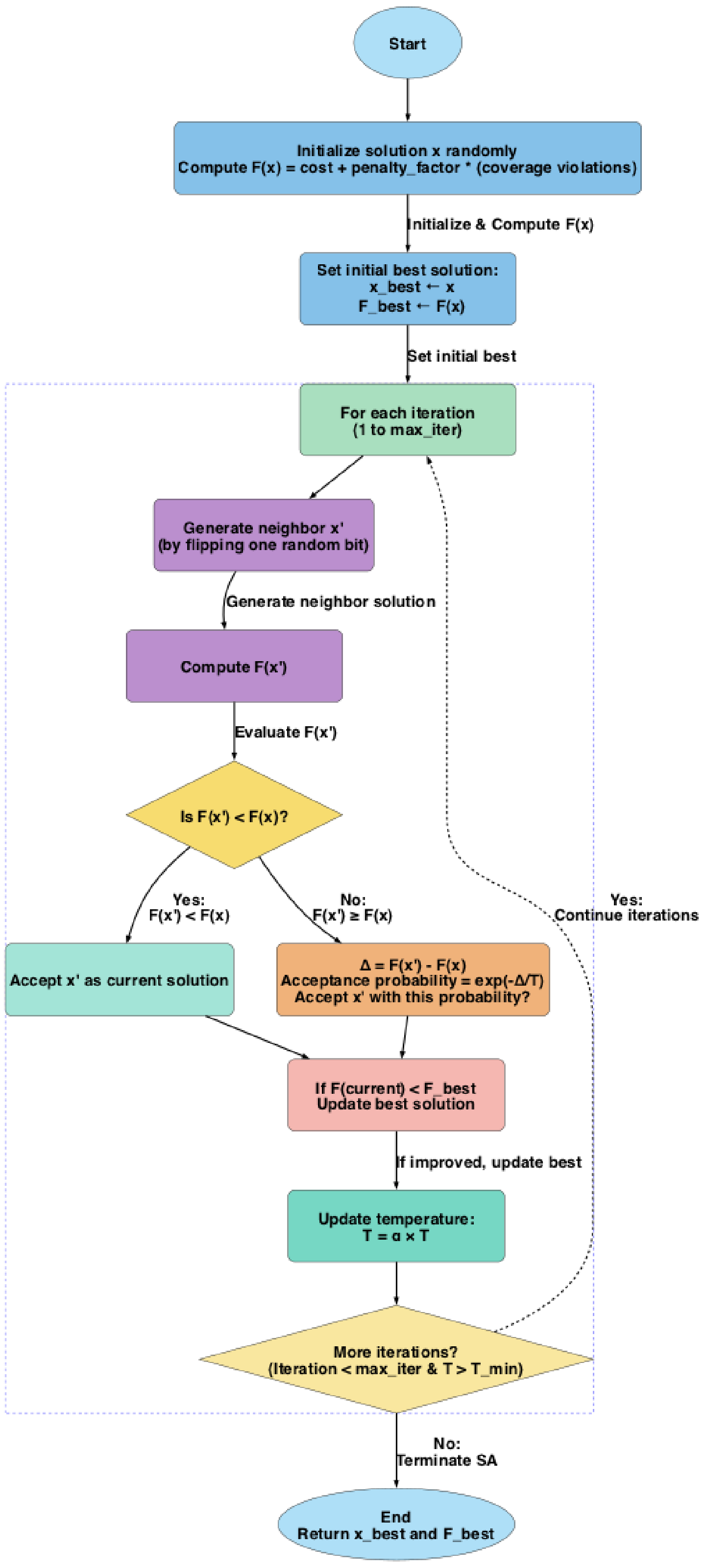

Figure 1: Flowchart of the Simulated Annealing Algorithm for the Set Partitioning Problem.

2.2 Standard Binary Genetic Algorithm

The Standard Binary Genetic Algorithm is a population-based technique that evolves a set of candidate solutions using selection, crossover, and mutation operators. It utilises tournament selection to choose parents, applies one-point crossover to combine their genetic material, and uses bit-flip mutation to introduce variability. A penalty function is employed to combine the cost of a solution with constraint violations, ensuring that feasibility is encouraged over generations. This method is well-suited for problems with discrete binary representations and offers robustness through its stochastic operations.

2.2.1 Pseudocode for Standard BGA

The pseudocode below outlines the Standard Binary Genetic Algorithm for the SPP, detailing population initialisation, tournament selection, crossover, mutation, and population update.

Algorithm 2 StandardBGA(*popSize*, *cxRate*, *mutRate*, *maxGens*, *penaltyFactor*, *tournK*)

```
1: Initialise:
2: $P \leftarrow$ RandomPopulation($popSize$)    ▷ Generate popSize random binary solutions
3: EvaluateFitness($P$, $penaltyFactor$)    ▷ Compute cost + penalty for each individual
4: $(xBest, F_{\text{best}}) \leftarrow$ BestSolution($P$)    ▷ Track best overall
5: for $g = 1 \to maxGens$ do
6:     $Q \leftarrow \varnothing$    ▷ Offspring population
7:     while $|Q| < popSize$ do
8:         $p_1 \leftarrow$ TournamentSelect($P$, $tournK$)    ▷ Select parent 1
9:         $p_2 \leftarrow$ TournamentSelect($P$, $tournK$)    ▷ Select parent 2
10:        if rand() $< cxRate$ then
11:            $(c_1, c_2) \leftarrow$ OnePointCrossover($p_1, p_2$)    ▷ Split at random point and swap tails
12:        else
13:            $c_1 \leftarrow p_1, \quad c_2 \leftarrow p_2$    ▷ No crossover; children are copies
14:        Mutate($c_1$, $mutRate$)
15:        Mutate($c_2$, $mutRate$)    ▷ Bit-flip mutation with prob = mutRate
16:        EvaluateFitness($\{c_1, c_2\}$, $penaltyFactor$)    ▷ Compute cost + penalty of each child
17:        $Q \leftarrow Q \cup \{c_1, c_2\}$
18:    $P \leftarrow Q$    ▷ Replace old population with offspring
19:    $(xCurrBest, F_{\text{currBest}}) \leftarrow$ BestSolution($P$)
20:    if $F_{\text{currBest}} < F_{\text{best}}$ then
21:        $xBest \leftarrow xCurrBest, \quad F_{\text{best}} \leftarrow F_{\text{currBest}}$    ▷ Update global best if improved
22: return $(xBest, F_{\text{best}})$    ▷ Best solution found
```

2.2.2 Flowchart for Standard BGA

The flowchart provides a visual overview of the Standard BGA's iterative process, from offspring generation to re-ranking and best solution update.

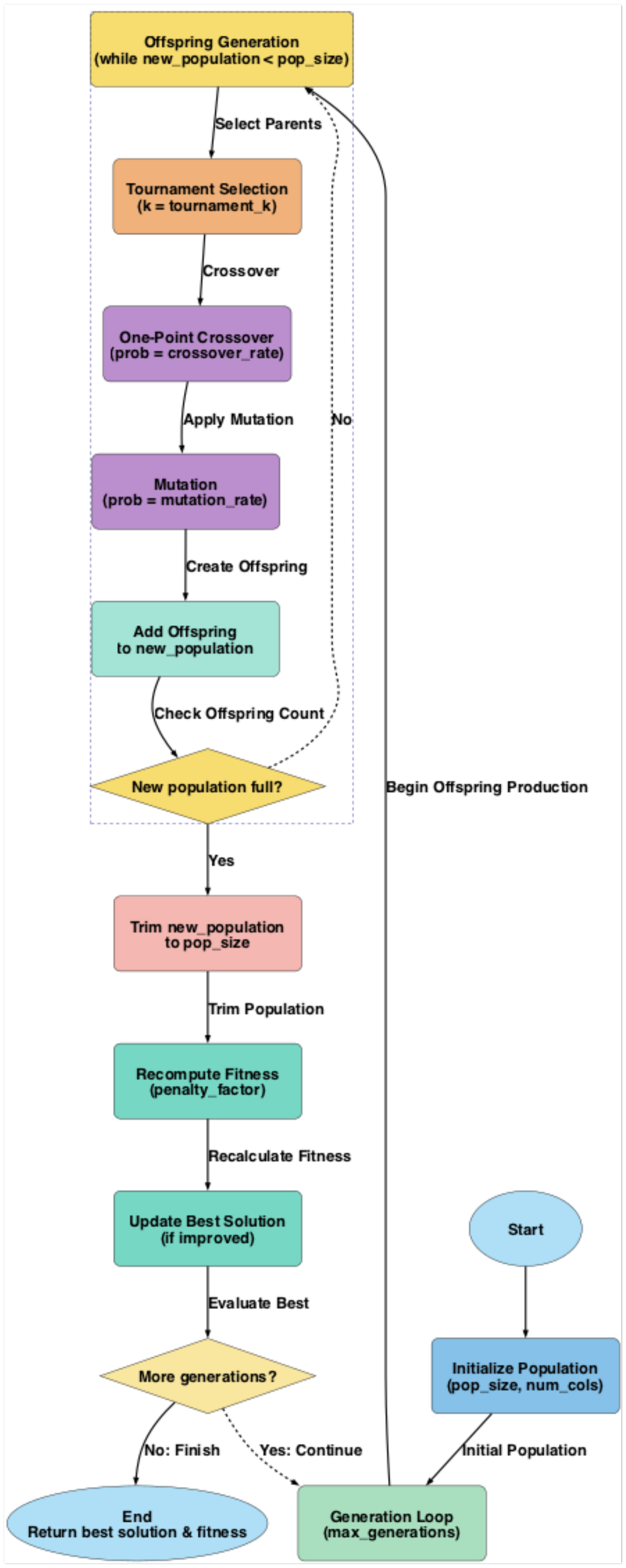

Figure 2: Flowchart of the Standard Binary Genetic Algorithm for the Set Partitioning Problem.

2.3 Improved Binary Genetic Algorithm

The Improved Binary Genetic Algorithm extends the standard Binary Genetic Algorithm by integrating several problem-specific enhancements. It begins with a pseudo–random initialisation that aims to reduce over–coverage in the initial population. The algorithm then incorporates a stochastic ranking procedure that probabilistically balances cost and constraint violation, allowing for a more flexible handling of infeasible solutions. Adaptive mutation is employed to reintroduce promising genetic material, and a DROP/ADD heuristic improvement operator repairs infeasible solutions. These enhancements, inspired by Chu & Beasley [1] and Runarsson & Yao [2], make the Improved BGA particularly robust for the SPP. Overall, this algorithm offers a more sophisticated search mechanism by blending global search operators with domain-specific repairs.

2.3.1 Pseudocode for Improved BGA

The pseudocode below describes the Improved Binary Genetic Algorithm for the SPP, integrating pseudo–random initialisation, stochastic ranking, adaptive mutation, and heuristic repair.

Algorithm 3 ImprovedBGA(*popSize*, *maxGens*, p_{stoch}, ...)

```
1:  Initialise:
2:  $P \leftarrow$ PseudoRandomInit($popSize$)            ▷ Chu & Beasley [1] for partial coverage
3:  EvaluateCostUnfitness($P$)            ▷ For each individual, compute (cost, violations)
4:  for $g = 1 \to maxGens$ do
5:      StochasticRankSort($P$, $p_{\text{stoch}}$)            ▷ Runarsson & Yao [2] sort by cost vs. unfitness
6:      $O \leftarrow \varnothing$            ▷ Offspring population
7:      while $|O| < popSize$ do
8:          $(p_1, p_2) \leftarrow$ SelectParents($P$)            ▷ Randomly pick two parents
9:          $(c_1, c_2) \leftarrow$ UniformCrossover($p_1, p_2$)            ▷ Exchange bits at random (50% each)
10:         AdaptiveMutation($c_1, P$)            ▷ Bit-flip mutation + forced coverage
11:         AdaptiveMutation($c_2, P$)
12:         HeuristicImprove($c_1$)            ▷ Repair via DROP/ADD (Chu & Beasley [1])
13:         HeuristicImprove($c_2$)
14:         EvaluateCostUnfitness($\{c_1, c_2\}$)
15:         $O \leftarrow O \cup \{c_1, c_2\}$
16:     $C \leftarrow P \cup O$            ▷ Combine parents & offspring
17:     StochasticRankSort($C$, $p_{\text{stoch}}$)            ▷ Re-sort combined pop
18:     $P \leftarrow$ Top($C$, $popSize$)            ▷ Keep the best $popSize$
19: return BestFeasible($P$)            ▷ Return best feasible (lowest cost, unfitness=0)
```

2.3.2 Flowchart for Improved BGA

The flowchart visually maps the Enhanced BGA's process, clearly highlighting its advanced offspring production and repair mechanisms.

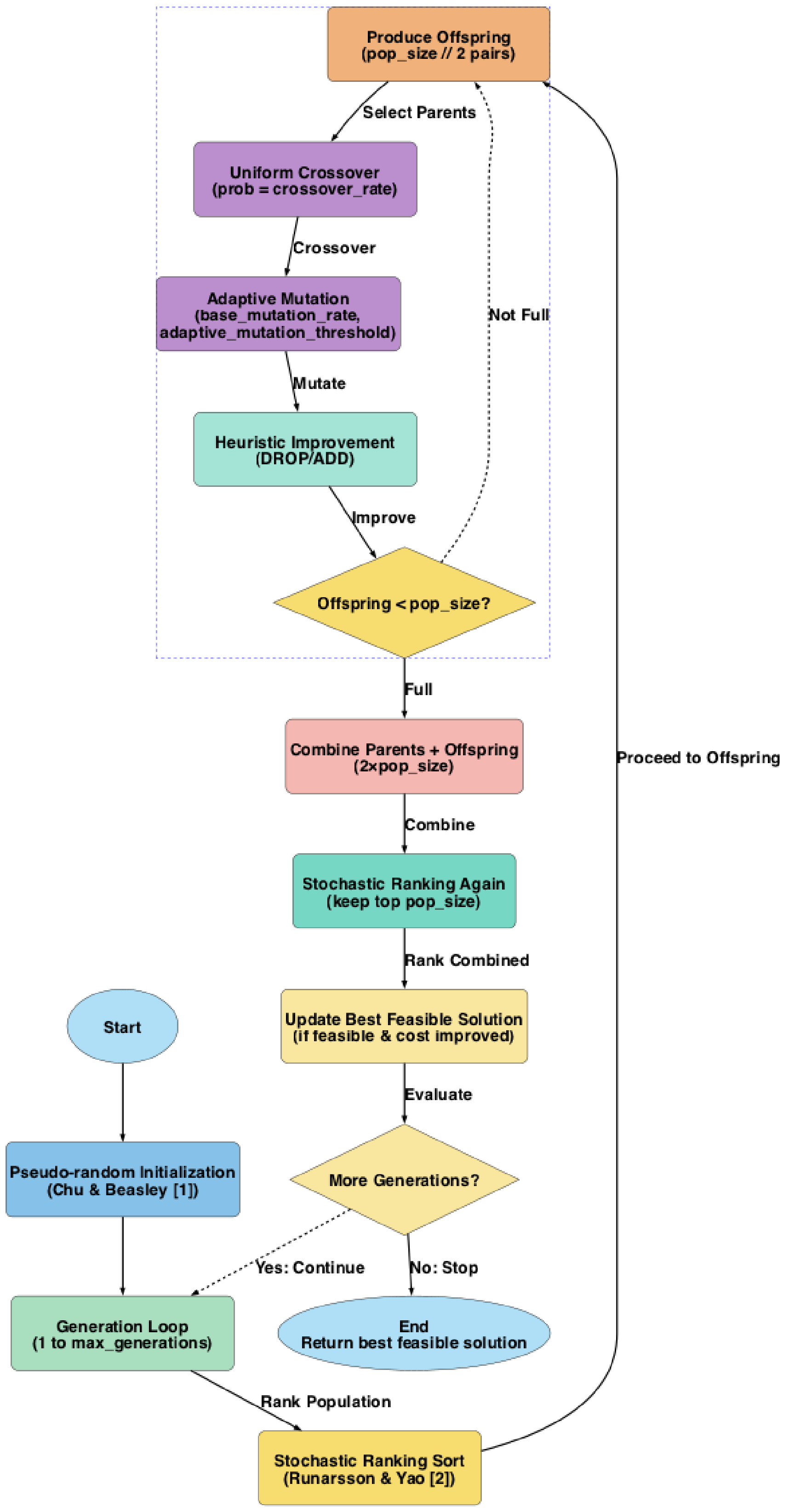

Figure 3: Flowchart of the Improved Binary Genetic Algorithm for the Set Partitioning Problem.

3 Benchmark Results

In this section, we present the results obtained by our three algorithms—Simulated Annealing (SA), Standard BGA, and Improved BGA—on three different problem sets. Each subsection provides a table summarising performance metrics across 30 independent runs, focusing on feasibility and overall solution quality. In particular, we report the rate at which each method produces a feasible solution (*Feas. Rate*), best/worst feasible cost, average feasible cost (with standard deviation), and average execution time (with standard deviation).

3.1 Problem Set `sppnw41`

We tested each algorithm for 30 independent runs on the `sppnw41` problem instance. Table 3.1 shows the feasibility rate, along with best, worst, and mean feasible costs (plus standard deviation), as well as average run time (plus standard deviation) for each algorithm. All results are based on runs that reached feasibility (zero coverage violations). A brief description of the hyperparameters used in each method can be found below.

Algorithm	Feas. Rate	Best Cost	Worst Cost	Mean Cost	Std. Dev.	Time (s) $\pm$ SD
SA	12/30	18,066	32,730	23,344	4,002	0.082 $\pm$ 0.0008
Standard BGA	21/30	12,678	28,269	20,188	3,736	1.94 $\pm$ 0.07
Improved BGA	30/30	11,307	13,395	12,177	658	2.65 $\pm$ 0.05

Table 1: Performance on `sppnw41` (30 runs). Feasible solutions have zero coverage violations.

Hyperparameters.

- **Simulated Annealing (SA).** Initial temperature = 2000.0, cooling factor $\alpha = 0.99$, maximum iterations = $10,000$, penalty factor = 50000.
- **Standard BGA.** Population size = 250, crossover rate = 0.8, mutation rate = 0.003, max generations = 500, penalty factor = 4800.0, tournament size = 4.
- **Improved BGA.** Population size = 50, max generations = 200, crossover rate = 0.85, base mutation rate = 0.03, stochastic rank $p = 0.45$, adaptive mutation threshold = 0.2, adaptive mutation count = 2.

As shown, the Improved BGA achieves a 100% feasibility rate and outperforms the other algorithms in terms of average feasible cost for this problem set, albeit with a slightly longer average run time.

3.2 Problem Set `sppnw42`

We tested each algorithm for 30 independent runs on the `sppnw42` problem instance. Table 3.2 shows the feasibility rate, along with best, worst, and mean feasible costs (plus standard deviation), as well as average run time (plus standard deviation) for each algorithm. All results are based on runs that reached feasibility (zero coverage violations). A brief description of the hyperparameters used in each method can be found below.

Algorithm	Feas. Rate	Best Cost	Worst Cost	Mean Cost	Std. Dev.	Time (s) ± SD
SA	1/30	14,222	14,222	14,222	0	0.608 ± 0.000
Standard BGA	0/30	—	—	—	—	—
Improved BGA	30/30	7,666	8,666	7,927	284	19.28 ± 0.78

Table 2: Performance on `sppnw42` (30 runs). Feasible solutions have zero coverage violations.

Hyperparameters.

- **Simulated Annealing (SA).** Initial temperature = 5000.0, cooling factor $\alpha = 0.99$, max iterations = 10,000, penalty factor = 100,000.
- **Standard BGA.** Population size = 400, crossover rate = 0.9, mutation rate = 0.003, max generations = 1000, penalty factor = 10,000, tournament size = 5.
- **Improved BGA.** Population size = 30, max generations = 400, crossover rate = 0.85, base mutation rate = 0.03, stochastic rank $p = 0.45$, adaptive mutation threshold = 0.25, adaptive mutation count = 2.

As shown, the Standard BGA failed to reach feasibility in any run, while Simulated Annealing found a valid solution only once. By contrast, the Improved BGA maintained a 100% feasibility rate and delivered competitive costs consistently.

3.3 Problem Set `sppnw43`

We tested each algorithm for 30 independent runs on the `sppnw43` problem instance. Table 3.3 shows the feasibility rate, along with best, worst, and mean feasible costs (plus standard deviation), as well as average run time (plus standard deviation) for each algorithm. All results are based on runs that reached feasibility (zero coverage violations). A brief description of the hyperparameters used in each method can be found below.

Algorithm	Feas. Rate	Best Cost	Worst Cost	Mean Cost	Std. Dev.	Time (s) ± SD
SA	5/30	11,996	17,614	14,600	2,100	0.57 ± 0.01
Standard BGA	0/30	—	—	—	—	—
Improved BGA	30/30	8,974	10,042	9,517	278	21.78 ± 0.47

Table 3: Performance on `sppnw43` (30 runs). Feasible solutions have zero coverage violations.

Hyperparameters.

- **Simulated Annealing (SA).** Initial temperature = 5000.0, cooling factor $\alpha = 0.99$, max iterations = 10,000, penalty factor = 100,000.
- **Standard BGA.** Population size = 400, crossover rate = 0.9, mutation rate = 0.003, max generations = 1000, penalty factor = 10,000, tournament size = 5.
- **Improved BGA.** Population size = 30, max generations = 400, crossover rate = 0.85, base mutation rate = 0.03, stochastic rank $p = 0.56$, adaptive mutation threshold = 0.25, adaptive mutation count = 2.

As indicated in Table 3.3, the Standard BGA again failed to produce any feasible solution, whereas Simulated Annealing identified a valid solution in only 16.7% of runs. By contrast, the Improved BGA achieved a 100% feasibility rate with a notably lower mean feasible cost.

4 Discussion and Comparison

5 Ranking Replacement vs. Stochastic Ranking

Both methods aim to balance cost and constraint satisfaction in the SPP, yet they differ in approach. The **Ranking Replacement** method (Chu & Beasley, 1998) partitions the population deterministically into four subgroups based on fitness (cost) and unfitness (constraint violation). A new solution replaces an individual from the first non-empty subgroup (starting with those worst in both criteria), thereby steadily improving the overall population. This approach offers a clear structure but can be rigid as the balance between cost and constraint violation may evolve during the search.

In contrast, the **Stochastic Ranking** method (Runarsson & Yao, 2000) uses a bubble-sort-like procedure where adjacent solutions are compared probabilistically—if at least one solution is infeasible, they are compared by cost with a set probability P (typically less than 0.5) and by unfitness otherwise. This allows for a more adaptive balance, enabling the search to explore infeasible regions as bridges between isolated feasible areas without rigid subgroup thresholds.

In summary, while both methods share the goal of guiding the search toward feasible, high-quality solutions, Ranking Replacement enforces a strict hierarchical structure, whereas Stochastic Ranking offers a flexible, adaptive mechanism that reduces the need for extensive parameter tuning.

6 Conclusion

This report has presented three advanced algorithms for solving airline crew scheduling problems: Simulated Annealing, a Standard Binary Genetic Algorithm, and an Improved Binary Genetic Algorithm incorporating problem-specific enhancements. Detailed, moderately sized flowcharts and refined pseudocode have been provided to elucidate each method's internal workings. Benchmark results and an in-depth discussion will be appended once experiments are complete. Furthermore, a comparative discussion on constraint-handling—contrasting ranking replacement and stochastic ranking—has been included, underscoring the adaptive advantages of the latter.

References

[1] P. C. Chu and J. E. Beasley, *Constraint Handling in Genetic Algorithms: The Set Partitioning Problem*, Journal of Heuristics, 11:323–357, 1998.

[2] T. P. Runarsson and X. Yao, *Stochastic Ranking for Constrained Evolutionary Optimisation*, IEEE Transactions on Evolutionary Computation, 4(3):284–294, 2000.