Programming Project 2

Giorgio Bertone, Jura Hostic

December 2024

1 Task 1

1.1 Genetic Algorithm

The GA framework operates on a population of candidate solutions, represented as permutations of graph vertices V, to minimize edge crossing costs while satisfying precedence constraints. The fitness function incorporates a penalty mechanism to discourage constraint violations, and ensure feasible solutions are favored.

Parameters

- **Population Size**: Number of individuals in each generation.
- **Generations**: Total number of generations to evolve.
- **Elite Size**: Number of top individuals preserved unaltered (elitism).
- **Tournament Size**: Size of subsets for tournament-based parent selection.
- **Mutation Rate**: Probability of applying mutation to offspring.
- **Crossover Rate**: Probability of applying the crossover operator.
- **Constraint Penalty**: Penalty weight for constraint violations in fitness evaluation.

Fitness Function

The fitness function combines the objective function (computed using *cost_function_bit*) and a penalty term proportional to the number of violated constraints to balance minimization of edge crossings with adherence to problem constraints.

Genetic Operators

1. **Selection**: A tournament selection mechanism is used to choose parents, favoring individuals with higher fitness.
2. **Crossover**: The Order Crossover (OX) operator generates offspring by preserving subsequences from one parent while maintaining valid permutations.
3. **Mutation**: Swap mutation, involving a variable number of element swaps, is applied to introduce diversity.
4. **Repair**: A repair mechanism ensures offspring are feasible by adjusting orderings to satisfy constraints when necessary.

Algorithm

- **Initialization**: The initial population is generated randomly and the individuals representing invalid solutions are repaired to ensure feasibility.
- **Evaluation**: Fitness scores are calculated for each individual in the population.
- **Selection and Reproduction**: Parents are selected through tournament selection, and offspring are produced using crossover, mutation, and repair.
- **Elitism**: The best-performing individuals are carried forward unchanged.
- **Iteration**: The process is repeated for a predefined number of generations, recording fitness statistics at each step.

1.2 Ant Colony Optimization

The MaxMin Ant System (MMAS) is a variation of the ACO metaheuristic, that controls the maximum and minimum pheromone amounts on each trail in order to avoid stagnation. Indeed, it has been shown empirically that MMAS strikes a good balance between intensification (exploiting good solutions) and diversification (exploring new regions). The MWCCP is modeled as a graph optimization problem, where ants traverse the solution space guided by pheromone trails and heuristic information.

Parameters

- **Alpha (α)**: Controls the influence of pheromones on ant decision-making; higher values promote exploitation.
- **Beta (β)**: Controls the importance of heuristic information; higher values favor heuristic-driven exploration.

- **Evaporation Rate** (ρ): Regulates pheromone decay; higher values reduce old pheromone influence more aggressively, enhancing exploration.
- **Ant Count**: Determines the number of ants per iteration; more ants increase diversity, but raise computational cost.
- **Iterations**: Sets the number of algorithm cycles; more iterations allow deeper exploration but require more time.
- **Tau Min** (τ_{min}): Limits minimum pheromone levels; prevents solution components from being ignored.
- **Tau Max** (τ_{max}): Caps maximum pheromone levels; ensures search diversity.

Components

- **Pheromone Matrix**: Initialized uniformly with high values to encourage exploration in early iterations.
- **Heuristic Information**: Derived from graph properties, such as vertex in-degree or edge weights, to prioritize promising candidates during solution construction.

Algorithm

1. **Initialisation**: The pheromone matrix and the heuristic information matrix are created as discussed above
2. **Solution Construction**: Each ant constructs an ordering of the vertices probabilistically based on pheromone intensity and heuristic attractiveness of each candidate vertex. The probabilities are calculated as a weighted combination of these factors, controlled by parameters that adjust their influence.
3. **Constraints Handling**: After constructing a solution, the algorithm verifies if it satisfies the constraints. If not a a repair mechanism rearranges it.
4. **Pheromone Update**: After all ants construct solutions, the pheromone levels are updated based on the global best solution. Evaporation is also applied to ensure pheromone decay over time, preventing premature convergence. Then the pheromone values are clipped.
5. **Dynamic Adjustment**: Upper and lower bounds for the pheromone levels are dynamically adjusted based on the best solution cost to maintain diversity and guide the search effectively.
6. **Iteration**: The algorithm iterates over multiple cycles, with each cycle involving solution construction, evaluation, and pheromone updates.

2 Task 2

2.1 SMAC3

For the hyperparameter optimization process we used SMAC3. SMAC3 is an open-source optimization framework design for challenging algorithm configuration and hyperparameter optimization problems. The framework uses Bayesian Optimization in combination with an aggresive racing mechanism to eficiently deterime which configurations are the best while balancing exploration and exploitation. The research paper is and the framework is available for use.

2.2 Final Configurations

The hyperparameter optimizations was done on medium tuning instances for time reasons. Smaller experiments were done on other instances with similar results. The SMAC3 framework was used.

For evaluating fitness, the algorithms were run on all medium tuning instances and the fitness was calculated using the cumulative cost and time using the following formula:

$$F = \begin{cases} C \cdot T & \text{if } T \leq 60\,\text{s}, \\ \frac{C \cdot T}{60} & \text{if } T > 60\,\text{s}. \end{cases}$$

The formula penalises longer running times which become too long on bigger instances, while considering all configurations with reasonable running times the same time wise to allow for limiting the comparison to the quality of the solutions. The time cutoff for penalisation was set to 60 seconds from testing the time differences of medium and large instances.

2.2.1 Genetic Algorithm

The genetic algorithm parameters was optimized using the following intervals:

Population Size:	$[20, 250]$
Generations:	$[10, 250]$
Elite Size:	$[0, 20]$
Tournament Size:	$[3, 30]$
Mutation Rate:	$[0.1, 0.6]$
Crossover Rate:	$[0.4, 1]$
Constraint Penalty:	$[10{,}000, 1{,}000{,}000]$

The tuning process was run for 250 trials. The best configuration found was:

Population Size:	40
Generations:	150
Elite Size:	13
Tournament Size:	25
Mutation Rate:	0.255
Crossover Rate:	0.79
Constraint Penalty:	213178

2.2.2 Ant Colony Optimization

The ant colony optimization parameters was optimized using the following intervals:

Alpha:	$[1.0, 2.0]$
Beta:	$[1.0, 2.0]$
Evaporation Rate:	$[0.1, 0.9]$
Ant Count:	$[20, 100]$
Iterations:	$[50, 250]$
Tau Min:	$[0.1, 1.0]$
Tau Max:	$[1.0, 10.0]$

The tuning process was run for 250 trials. The best configuration found was:

Alpha:	1.87
Beta:	1.07
Evaporation Rate:	0.69
Ant Count:	22
Iterations:	55
Tau Min:	0.51
Tau Max:	7.85

2.3 Influence of Parameters

The paramaters which were chosen during the optimization process strike a balance between solution quality and the time required to find it. The following graphs show the influence of different parameters on the solution quality and the time required to find it. In most of them, you can see that the parameter value which was chosen during the optimization process (Red dots in graphs) is often one of the best values for the cost and time trade-off. The graphs are cumulative cost and time over all medium tuning instances, with the parameter values being the same as the ones used in the final configuration except for the parameter being tested.

2.3.1 Genetic Algorithm

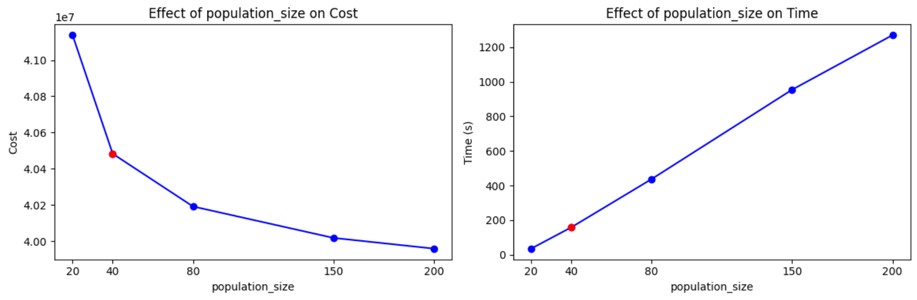

Figure 1: Influence of Population Size on Fitness

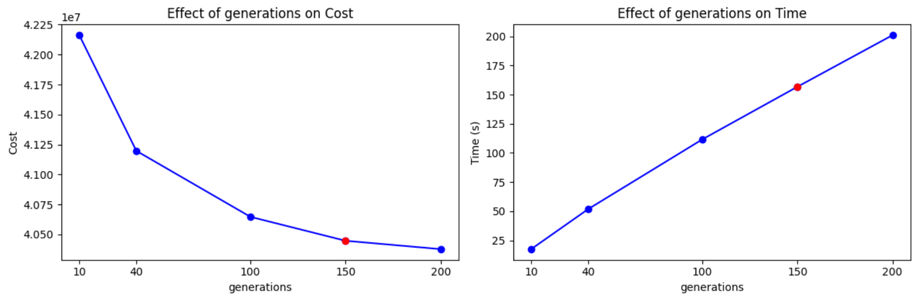

Figure 2: Influence of Generations on Fitness

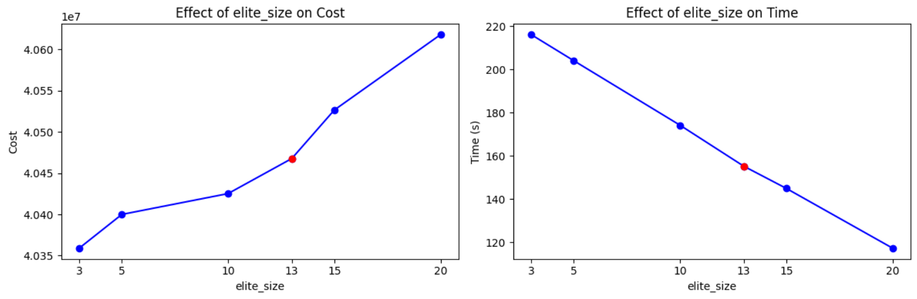

Figure 3: Influence of Elite Size on Fitness

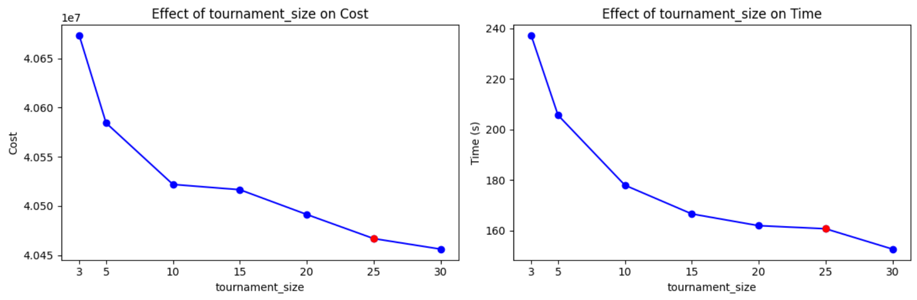

Figure 4: Influence of Tournament Size on Fitness

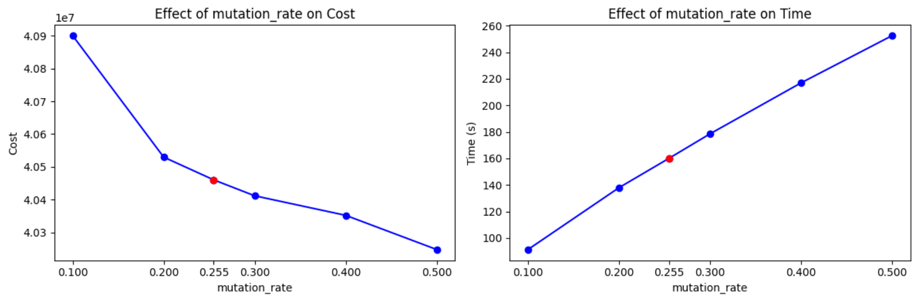

Figure 5: Influence of Mutation Rate on Fitness

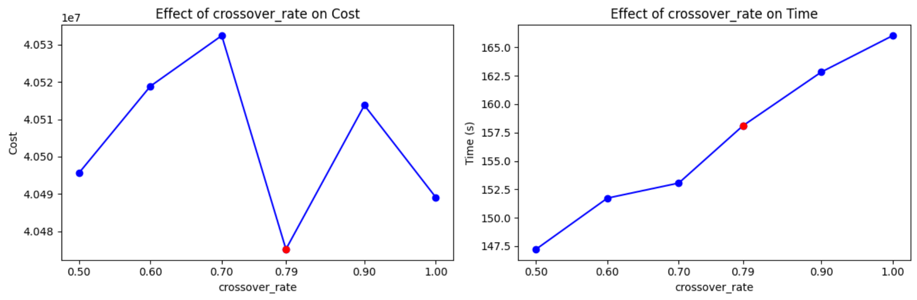

Figure 6: Influence of Crossover Rate on Fitness

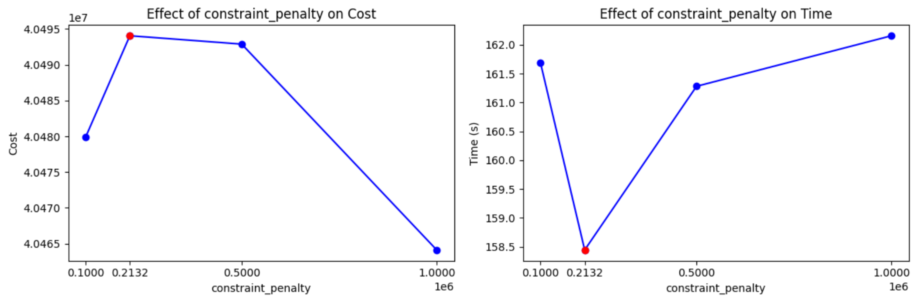

Figure 7: Influence of Constraint Penalty on Fitness

2.3.2 Ant Colony Optimization

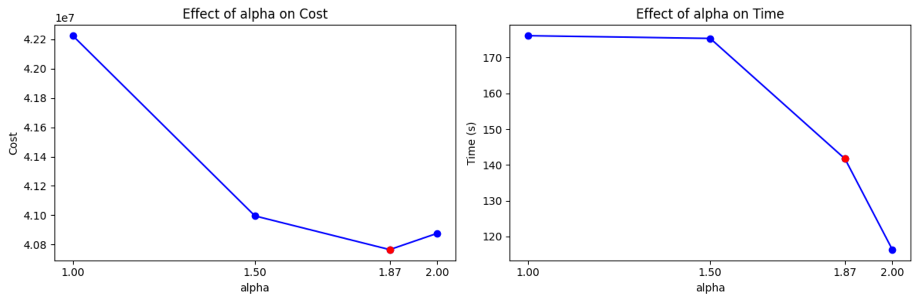

Figure 8: Influence of Alpha on Fitness

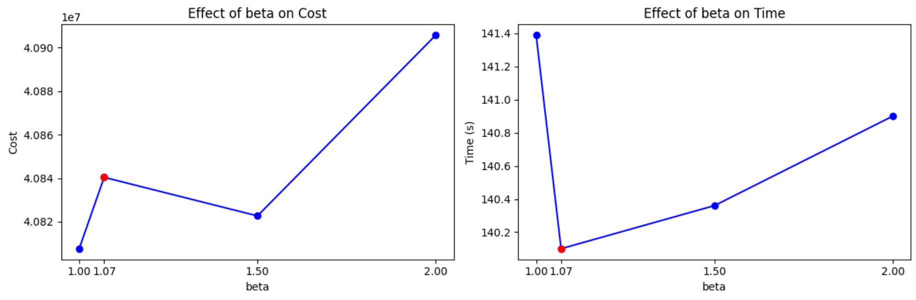

Figure 9: Influence of Beta on Fitness

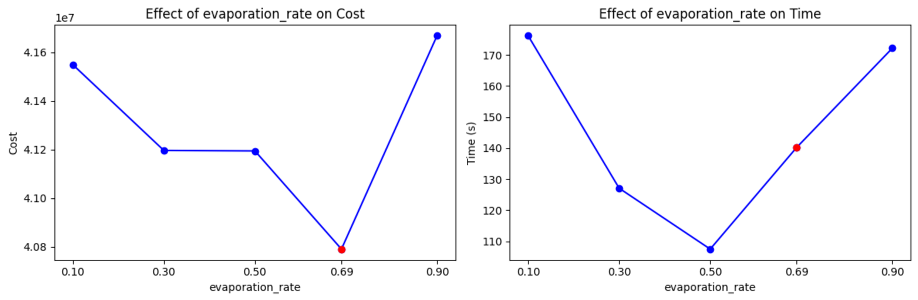

Figure 10: Influence of Evaporation Rate on Fitness

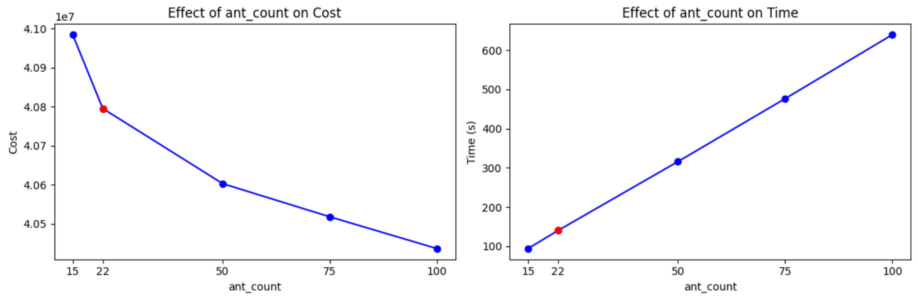

Figure 11: Influence of Ant Count on Fitness

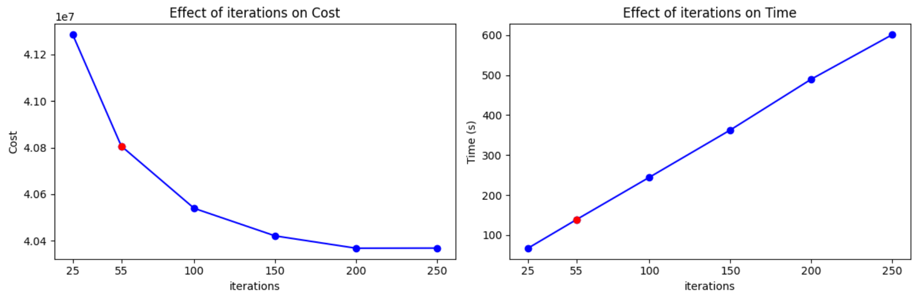

Figure 12: Influence of Iterations on Fitness

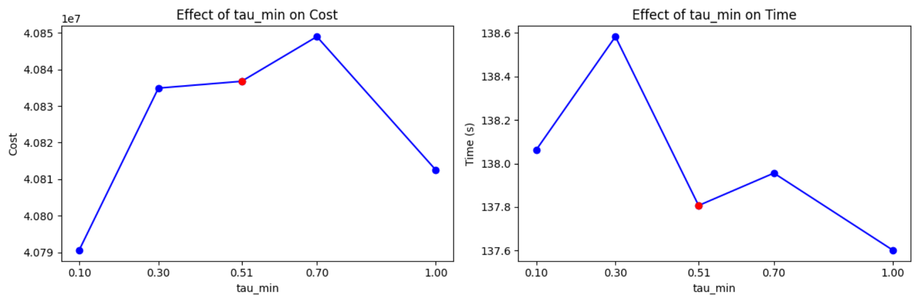

Figure 13: Influence of Tau Min on Fitness

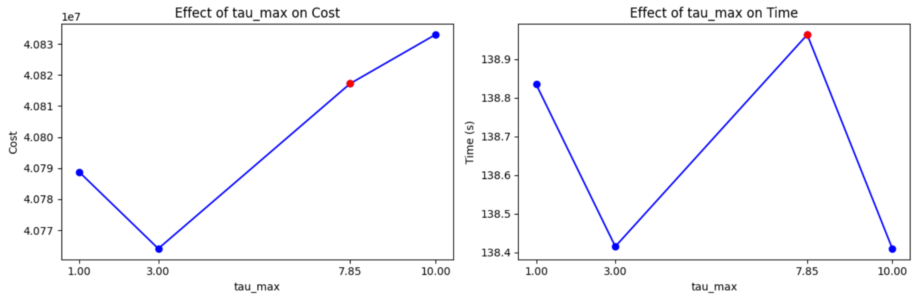

Figure 14: Influence of Tau Max on Fitness

2.4 Bottlenecks

2.4.1 Genetic Algorithm

In the genetic algorithm, the cost function is almost exclusively the bottleneck. About 96% of the time is spent on the cost function, with other functions like the repair mechanism and generating the new generation taking up most of the remaining time. The runtime could be greatly improved by optimizing the cost function.

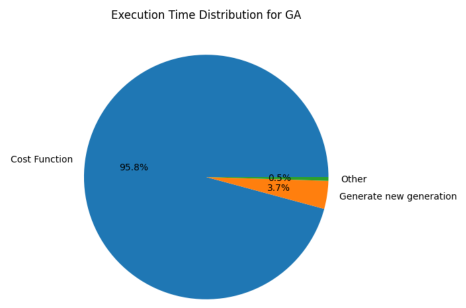

Figure 15: Bottleneck Analysis for GA (large instances)

2.4.2 Ant Colony Optimization

In the ant colony optimization algorithm, the cost function is by far the biggest bottleneck. About 81% of the time is spent on the cost function. The next biggest bottleneck is the solution construction, which takes about 18% of the time including all of its methods like the repair and validation methods. The runtime could be greatly improved by optimizing the cost function.

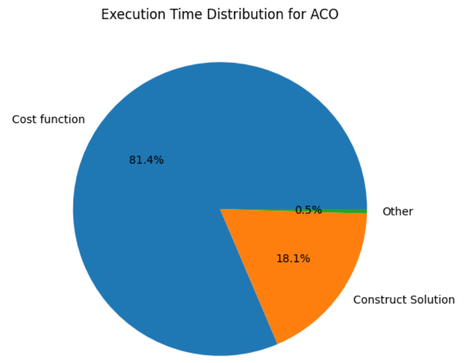

Figure 16: Bottleneck Analysis for ACO (medium-large instances)

3 Task 3

To test whether there is a significant difference between the performance of the two algorithms on the test instances, we applied the tuned algorithms 30 times on each test instance for small and medium instances. For medium-large and large instances, we ran the algorithms 5 times each due to time constraints. For each instance, we recorded the best performance and compared the algorithms using a statistical test for paired samples since the objective function values are determined on the same instances.

3.1 Small

First, we tested whether the difference in performance between the two algorithms follows a normal distribution.

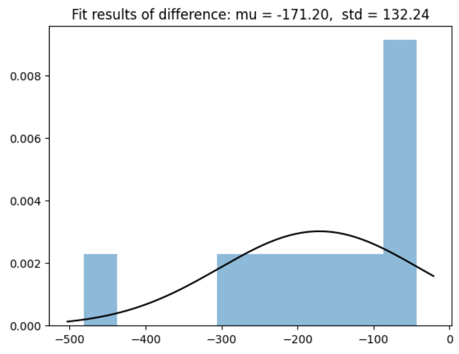

Figure 17: Performance difference on small instances

Since the distribution is not normal, we used the **Wilcoxon test**, a non-parametric test. We performed a two-sided test with the null hypothesis and alternative hypothesis defined as:

$$H_0 : \theta_{GA} = \theta_{ACO}$$
$$H_1 : \theta_{GA} \neq \theta_{ACO}$$

The test result was:

```
Test Result:
p-value = 0.0020
H0 can be rejected on a level of significance of 0.05
```

This indicates statistical evidence of a difference between the two algorithms.

To interpret the direction of the effect, since we can not repeat a test changing alternative as this would be considered p-hacking, we looked at the mean and median of the difference between the performances as the test just told us the results are statistically significant.

We see that:

```
    Mean Difference: -171.2
    Median Difference: -126.5
```

The negative mean and median difference indicate that the objective function values of the solutions found by the ACO algorithm tend to be higher, and considering we want to minimize this value we can conclude on the small instances the GA algorithm performs better.

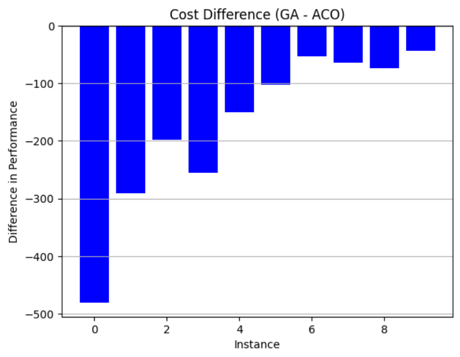

Figure 18: Cost difference between solutions found by GA and ACO for each small instance

We also analyzed the standard deviation of the solutions and runtime differences:

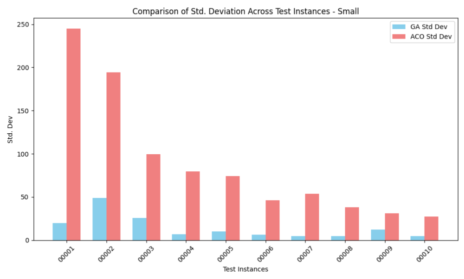

Figure 19: Standard deviation of solution costs for small instances

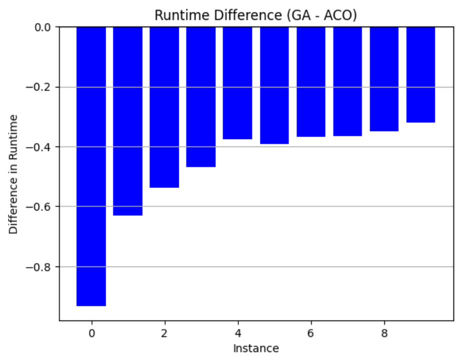

Figure 20: Runtime difference between GA and ACO for small instances

3.2 Medium

The same process was applied to medium instances. The performance difference was tested for normality:

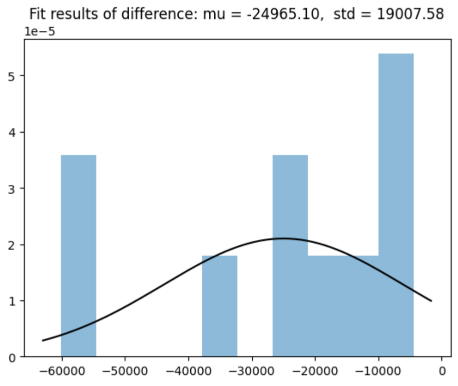

Figure 21: Performance difference on medium instances

Again, the distribution was not normal, so we applied the Wilcoxon test:

```
Test Result:
p-value = 0.0020
H0 can be rejected on a level of significance of 0.05.
```

The mean and median differences were:

```
Mean Difference: -24965.1
Median Difference: -20181.0
```

These results indicate that ACO performs worse than GA for medium instances as well.

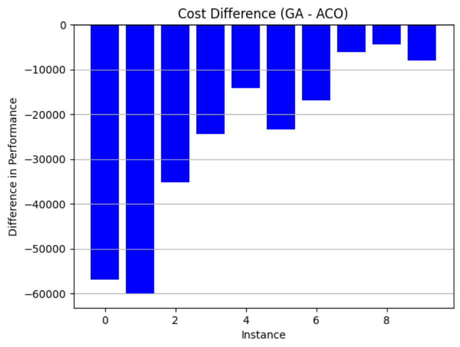

Figure 22: Cost difference between solutions found by GA and ACO for medium instances

Standard deviation and runtime analyses were also performed:

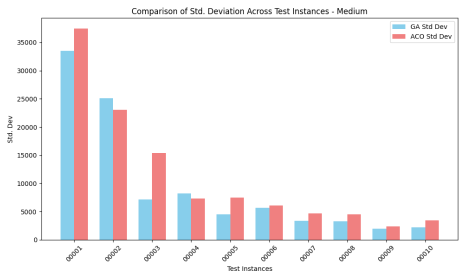

Figure 23: Standard deviation of solution costs for medium instances

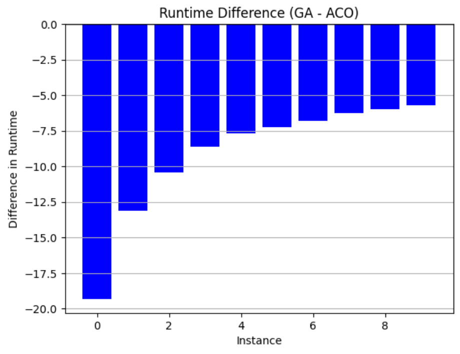

Figure 24: Runtime difference of GA and ACO on medium instances

3.3 Medium-Large

We repeat once again the same process for the medium-large test instances.

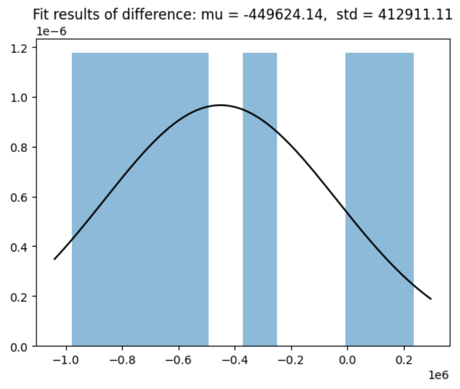

Figure 25: Performance difference on medium-large instances

Again this doesn't look like a normal distribution so we will use the Wilcoxon test, from which we get:

```
Test Result: p-value = 0.0781
H0 cannot be rejected on a level of significance of 0.05.
```

This time, we cannot reject the null hypothesis at level of significance of 0.05, but we can do it for significance 0.1. We also computed the negative mean and median to have more information about the direction of the effect:

```
Mean Difference: -449624.14
Median Difference: -533586.0
```

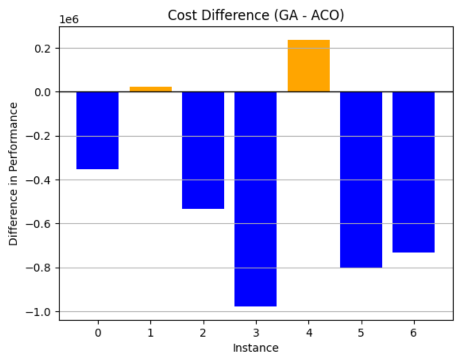

Figure 26: Cost difference on medium-large instances

This suggests that ACO performs worse than GA on medium-large instances at a 0.1 significance level.

As before, we also include the standard deviation of the solutions.

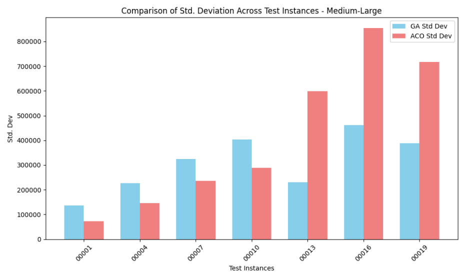

Figure 27: Standard Deviation of solutions on medium-large instances

We also plot the runtime difference between the two algorithms.

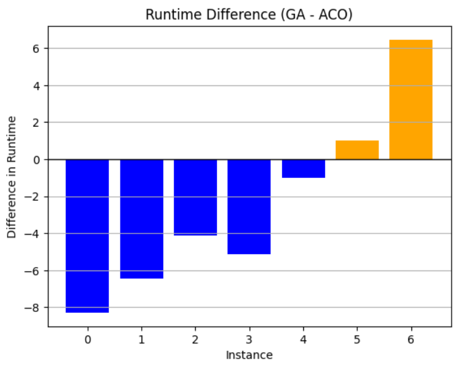

Figure 28: Runtime difference on medium-large instances

3.4 Large

Finally, we repeat the same process for large test instances.

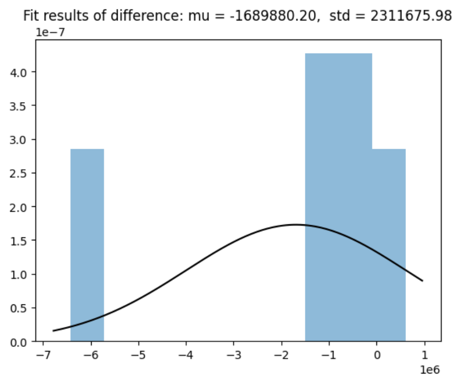

Figure 29: Performance difference on large instances

Once again this does not look like a normal distribution so we will use the Wilcoxon test, from which we get:

```
Test Result:
p-value = 0.0195
H0 can be rejected on a level of significance of 0.05.
```

The mean and median differences were:

```
Mean Difference: -1689880.2
Median Difference: -859409.0
```

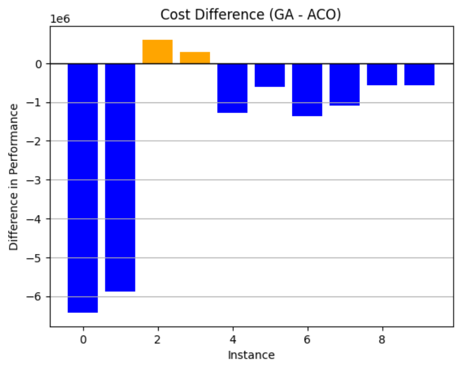

Figure 30: Cost difference for large instances

We can conclude that also on large instances, ACO performs worse than GA with significance of 0.05. As before, we also include the standard deviation of the solutions.

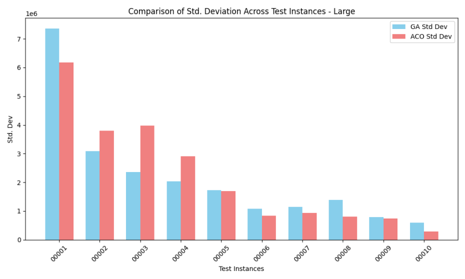

Figure 31: Standard Deviation of solutions on large instances

We also compare the runtimes.

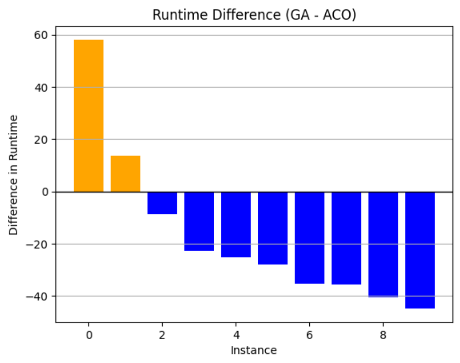

Figure 32: Runtime difference on large instances

4 Task 4

We now run the GA, which turned out to be better from the tests, on the test instances in order to do some experiments and get more insights into the performance of the algorithm. In particular, we include plots that show the quality of the solution achieved over time.

For small instances, we see a rapid early improvement and a quick convergence to a high-quality solution. There seems to be a good balance between exploration and exploitation since a good exploration is indicated by the initial variability in the cost values and large improvements, while the exploitation occurs as the curve flattens, focusing on refining solutions. Moreover, the gap between best and average solutions seems to narrow pretty quickly, indicating convergence (however, this could also be a sign of premature convergence and further investigation could be required before applying this algorithm to a real world scenario).

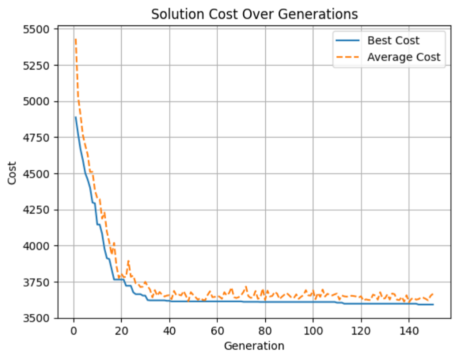

Figure 33: Average and best solution cost over time - *inst_50_4_00005*

For medium, medium-large and large instances, this behavior is still visible but less pronounced (in particular for the large instances that would seem to benefit from some more iterations). However, the algorithm still seems to converge to a high quality solution fast and the curves exhibit, overall, the same characteristics as before.

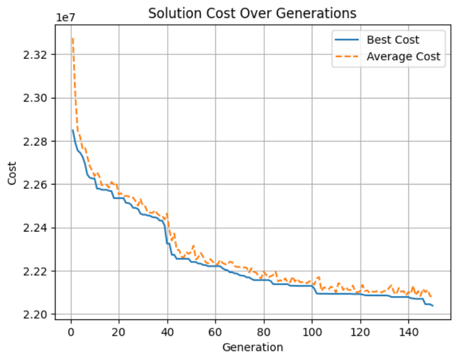

Figure 34: Average and best solution cost over time - *inst_200_20_00001*

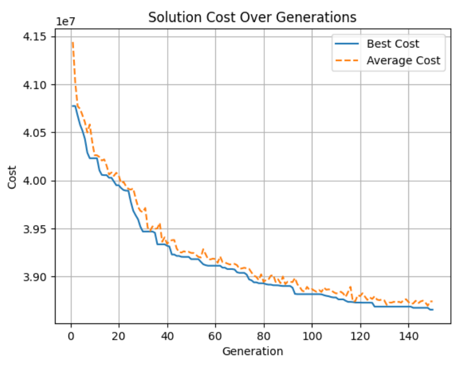

Figure 35: Average and best solution cost over time - *inst_500_40_00001*

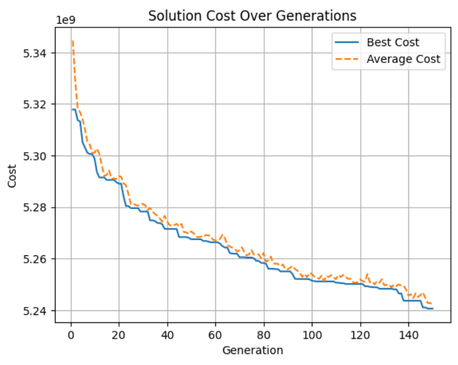

Figure 36: Average and best solution cost over time - *inst_1000_60_00002*

Overall, the curves show:

- Rapid decrease in cost early on.
- Gradual flattening of the curve as it approaches the global or near-global optimum.
- Narrowing gap between best and average cost

We can conclude the algorithm is effectively balancing exploration and exploitation, and parameters are likely well-tuned.

4.1 Scalability Testing

The plots do not suggest particular scalability issues. In particular, the runtime seems to increase quadratically with size.

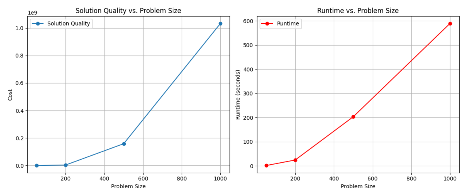

Figure 37: Scalability Testing on Different Instance Sizes

We tested our hypothesis by fitting a linear, a quadratic and an exponential model to our runtime data and, indeed, we got the following results:

```
MSE (Linear): 1641.45
MSE (Quadratic): 165.48
MSE (Exponential): 1492.80

R-squared (Linear): 0.971
R-squared (Quadratic): 0.997
R-squared (Exponential): 0.974
```

4.2 Generations vs Solution Quality Trade-off

The value 150 found through parameter tuning seems reasonable since, as we can see from the plots below, we have diminishing returns after this point.

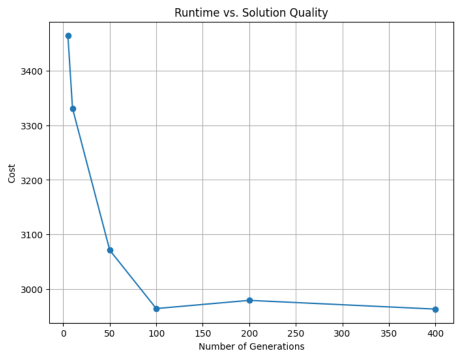

Figure 38: Generations vs Solution Quality - inst_50_4_00006

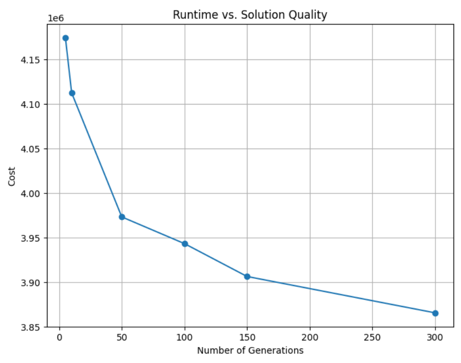

Figure 39: Generations vs Solution Quality - inst_200_20_00003

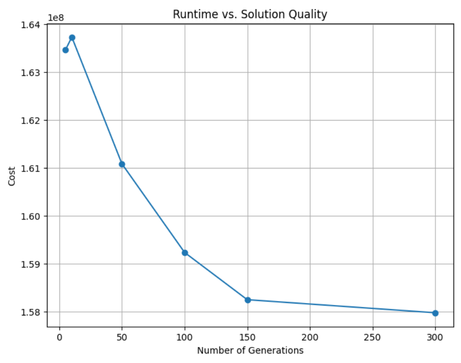

Figure 40: Generations vs Solution Quality - inst_500_40_00007