Master in Data Science and Advanced Analytics

NOVA Information Management School
Universidade Nova de Lisboa

Optimizing Wedding Seating Arrangements

Group R

Gaspar Pereira
Maria Cruz
Ricardo Pereira
Rita Wang

Computational Intelligence for Optimization
Inês Magessi, Leonardo Vanneschi

Code implementation at:
https://github.com/gpimenta42/CIFO_24_25

Spring Semester 2024-2025

Introduction

In the context of the Computational Intelligence for Optimization course we explored three algorithmic approaches to solve the Wedding Seating Problem, where the goal is to maximize the overall happiness of guests based on relationship scores that are predefined. Our primary focus was on implementing and evaluating different

configurations of the Genetic Algorithm and comparing its results against Simulated Annealing and Hill Climbing (Vanneschi and Silva, 2023). The code used for this project can be found in the GitHub repository.

1 Problem Definition

The goal of the Wedding Seating Optimization problem is to find the best seating arrangement for 64 guests across 8 tables, with 8 guests each, while maximizing the total happiness score - which is calculated based on the pairwise relationship between guests seated at the same table. Note that each guest must be assigned to exactly one table, and no guest can be left unassigned, while the order within the table and between tables does not matter.

A chart with the guests' relationships (Figure 1) as well as the assigned values for the different relationships (Table 1) can be found in the appendix.

2 Solution Implementation

To model a seating arrangement, we implemented the `Wedding_Solution` class, where each solution is represented as list of lists of shape 8×8, and each row corresponds to a table that contains exactly 8 guests (integers 1 to 64). We chose this structure because it simplifies the calculation of the happiness score for each table and offers easier readability and interpretation.

The happiness score for a given seating arrangement is calculated by the fitness function, which goes through each table and sums the relationship score between each unique pair of guests seated at that table, using the pairwise relationship matrix provided. And the total fitness score is the sum of the scores across all tables:

$$\sum_{t \in T} \sum_{\substack{i,j \in G_t \\ i \neq j}} \text{Relationship}(i, j)$$

- $T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8\}$, 8 sets of tables; ● $G_!$ = Guests seated at the same table t;
- $Relationship(i, j) =$ Relationship scores between guests i and j.

Note that for each table there can be $(8 \times 7)/2 = 28$ unique pairs of guests, given that $(i, j) = (j, i)$.

3 Search Space

The search space of this problem consists of all the valid ways to assign 64 guests to 8 tables of exactly 8 guests each, so the total number of possible seating arrangements is given by:

$$\frac{64!}{(8!)^8 \times 8!} \approx 4.51 \times 10^{47}$$

The number of possible arrangements is extremely large, reinforcing the need for an efficient optimization algorithm to explore the search space.

4 Optimization Algorithms

4.1 Genetic Algorithm

To implement the genetic algorithm, we optimized the code so that for each generation we only need to calculate the fitness of all individuals of the population once and pass the fitness values to the selection functions that are dependent on them. This was done because the fitness evaluation is the most time expensive operation in the algorithm. With this it is expected that the number of fitness evaluations will be:

$$\text{Number of fitness evaluations} = n_{\text{generations}} \times n_{\text{population}}$$

4.1.1 Selection Algorithms

For the selection algorithms we decided to explore Rank Selection and Tournament Selection, as these were presented in class but not applied. In rank-based selection, each individual is first ranked from best to worst based on fitness, and in a maximization problem such as our own, it means that the individual with worst fitness will

have rank 1, and the best will have rank N (Figure 2). The probability of being selected is computed using an exponential function: $e^{"\#\cdot(\&"')}$

$$P(selecting \quad = \frac{}{\sum^{N} \quad e^{-\lambda\cdot(N-j)}}, N = \qquad i)$$

$$Population$$

$$size\ )^{*1}$$

The exponential function allows us to control the selection pressure via the parameter λ, where higher values assign an exponentially greater probability to higher-ranked individuals (Figure 3). To perform the selection, we adapted the roulette wheel mechanism, where a random number is drawn from the uniform distribution $[0,1]$, and individuals are selected based on the cumulative sum of their selection probabilities.

For tournament selection, we chose to implement it by selecting k individuals with replacement, as this gives a chance for the worst individuals to be selected as well, which is not possible without replacement. The individual with the highest fitness value will be chosen from this group (Figure 4).

4.1.2 Crossover Operators

Taking the Classic Order crossover and Partially Mapped crossover, we adapted them to fit our solution representation, to not disrupt the table structures too much. For the custom Order crossover, the two crossover points will respect the table structure, meaning that the child will have n random (1 to 7) tables equal to one parent and the rest of the tables will be filled with the other parent, in the order they appear, excluding any guests already present. The order of the guests in each table is shuffled before and after filling the child to avoid positional bias (Figure 5).

In the adapted Partially Mapped Crossover, the mapping region will also respect the table structures, starting always in the beginning of a table and ending at the end of the same or a different table (length of the mapping region is random from 1 to 7). The rest of the table guests will be filled from the other parent, and if a guest is already present in the child, the mapping will be used to find the corresponding missing guest (Figure 6).

We tested the distribution of structure disruption (measured as the inverse of the proportion of pairs of guests preserved from the parents) and confirmed that our custom crossover functions are less disruptive than classic order and classic partially mapped crossover (Figure 10)

4.1.3 Mutation Operators

The mutation operators implemented were Swap, Inversion and Scramble mutations, and just like in crossover, we picked these to not overly disrupt the table representation. Firstly, Swap mutation works by randomly selecting two distinct tables and one guest from each table, and then it swaps their seats (Figure 7).

Both Inversion mutation and Scramble mutation need to flatten the representation into a one-dimensional list before applying the mutation, which is then converted back into the matrix representation in the end. The Inversion mutation randomly picks two different indices to define a segment to invert the order of the guests seated in it (Figure 8). On the other hand, Scramble mutation chooses a random number of guests, with a bias towards a smaller number, and randomly shuffles them (Figure 9).

We also measured the structure disruption associated with the mutation functions. We observed that swap mutation has a constant disruption (it always swaps two guests between two different tables). The inversion mutation and scramble mutation have a higher expected disruption, with scramble mutation having the highest expected disruption (Figure 11).

We later added a parameter (k) to the scramble mutation to control the size of the scramble. This parameter adjusts the weight assigned to each possible scramble size i:

$$w_i = \frac{1}{i^k}, \quad \text{for } i = 2, \ldots, 63$$

A higher value of k increases the denominator for larger scramble sizes, resulting in a lower probability of selecting large scrambles. This favours smaller scrambles and introduces less disruption. (Figure 12).

4.2 Simulated Annealing Algorithm

Our implementation of the simulated annealing algorithm was also optimized to only calculate the fitness of the random neighbours and caching the fitness of the current solution. With this we expect the number of fitness evaluations to be equal to:

$$\text{Number of fitness evaluations} = n_{\text{iterations}} \times L$$

L being the number of random neighbours the current solution will be compared to in the same iteration, with the same C value (denominator of the acceptance probability formula).

The Wedding Solution SA subclass has a new method to generate random neighbours to be compared with the current solution in their fitness. The random neighbour is generated by applying a mutation function to the current solution with mutation probability of 1. We applied two mutation functions as random neighbour generators for simulated annealing: swap mutation and scramble mutation.

4.3 Hill Climbing Algorithm

For the Hill Climbing we created the Wedding Solution HC subclass, which had a method to generate all the neighbours to the current solution through the swap mutation. Figure 13 has the representation of the classes and subclasses created, with associated methods.
To note that the number of fitness evaluations is equal to:

$$\text{Number of fitness evaluations} = n_{\text{iterations}} \times n_{\text{neighbors}}$$

And the number of neighbours using swap mutation is equal to:

$$\binom{8}{2} \times 8 \times 8 = 1792$$

We used swap mutation as the sole neighbourhood generator in hill climbing because it guarantees a fixed and manageable number of neighbours.

5 Algorithms Evaluation

To perform the evaluation of each algorithm we used grid search to find the best combination of hyperparameters. Each combination was evaluated with 30 repetitions, with a different random initial solution or population for each repetition, to analyse the statistical significance of the results when comparing the final best fitness for each algorithm.

For the top 10 performing combinations for each algorithm, we repeated the evaluation 100 times and with more iterations / generations to check for consistency. For this comparison we also used the top-1 configuration of other hyperparameters not present in the top 10 combinations (for example: if the top 10 combinations all had a swap mutation operator, we would use the top 1 combination for each other mutation operators used in grid search, to ensure diversity in the comparisons.

All the hyperparameters used in the grid search for each algorithm are presented in table 2. In total we ran 4,800 different configurations for GA, 68 for SA and 2 for HC.

For the first round of evaluations (30 repetitions), we used 5,000 fitness evaluations to identify the best hyperparameter combinations for GA and SA. For the top-performing combinations, as well as for HC, we used around 10,000 fitness evaluations (see the combination of hyperparameters that results in those fitness evaluations Table 3). We used the same number of fitness evaluations for all algorithms, to ensure a fair comparison. For HC we additionally ran with 100 iterations (179,200 fitness evaluations) to see if the algorithm would converge to a better solution.

The results from these evaluations were stored in the corresponding csv files, which were used to compare the performance of different algorithm configurations based on their fitness scores, by analysing the boxplots and assessing the overall variance and median value. When comparing key configurations we also applied nonparametric statistical tests, as normality is not assumed. To check the statistical significance in the difference between the independent groups of more than two, we used the Kruskal-Wallis test ($\alpha = 0.01$), and calculated η^2 to measure the effect size. In the case of significant differences, we followed up with Dunn's test (with a

significance level of $p < 0.01$) to identify which pairs of groups showed meaningful differences. As for the comparison of a pair of independent groups, we applied Mann-Whitney U test ($\alpha < 0.01$) and evaluated the effect size using rank-biserial correlation.

5.1 Genetic Algorithm Results

To evaluate the different configurations of the Genetic Algorithm, we performed aggregations, starting with grouping by mutation, crossover and selection operators (Figure 14). Swap mutation consistently outperformed the inversion and scramble, independent of the crossover or selection method, indicating it was the main contributor to performance. We also noticed that for both inversion and scramble mutation, using partially mapped crossover seemed to have a positive effect, which could be due to the fact that it is less disruptive to the table structure.

When aggregating only by mutation (Figure 15) we confirm the results observed previously, with swap mutation clearly outperforming other two, as well as showing lower variance than scramble, suggesting greater stability. Further analysis of the mutations grouped by probabilities (Figure 16), showed that higher mutation rates (≥ 0.5) improved performance across all mutation types, by introducing more diversity to the population. As for crossover operators (Figure 17), the difference in performance was minor, although partially mapped crossover performed slightly better, albeit with a slightly greater variance. Crossover probabilities (Figure 18) in the [0.8-1] range were beneficial to both crossover types. Given that these methods were adapted to better preserve the table structure, higher rates likely enhanced diversity without excessive disruption. However, when comparing maximum median values from mutation and crossover analysis, swap mutation has the most significant impact on performance, reinforcing the idea that effective mutation can contribute more to exploration than crossover, leading to better results.

Grouping by selection method, rank selection seemed to have achieved higher median fitness values and lower variance compared to tournament (Figure 19). However, tournament selection with a larger tournament size (15) outperformed all other configurations, with rank selection with $\lambda = 1$ coming second with close median values but showed higher variance (Figure 20). Despite rank selection presenting a better overall performance, having a larger tournament size is more effective in achieving higher performance.

The addition of elitism also had a positive impact (Figure 21), resulting in slightly higher median fitness value. In contrast, configurations without elitism showed greater variance, likely reflective of the effects of lower selection pressure.

Lastly, for the top configurations (check Table 4 for configuration numbering), when analysing fitness over generations (Figure 22), most of them performed similarly, with only two configurations underperforming noticeably. Elitism was present in 8/14 of the top configurations, suggesting it does offer a slight performance boost, as seen earlier. Swap mutation clearly dominated, appearing in all top 8 configurations, reinforcing its role as the most effective mutation operator. While partially mapped crossover was present in most top configurations, group-preserving order crossover achieved the highest final fitness value of 76000 when combined with rank selection ($\lambda = 10$). Which could suggest that although using partially mapped crossover may lead to more consistently higher performances, group preserving crossover can still outperform in the right setup. Consistent as well with results presented above, all top configurations used high mutation and crossover rates, supporting the idea that increased diversity helps to reach better performance. The same goes for higher rank and tournament parameter values.

Although the boxplot of the final runs (Figure 23) show that configuration 11 has the highest median value and lower variance, we believe that this is due to faster convergence and not better performance. This is evident when comparing its fitness per generation plots (Figure 24) against the top 3 best final fitness configurations (0,1 and 2), where we can see that it plateaus earlier and it noticeably underperforms against the others. It also shows a bumpier progression line, suggesting its instability in performance over the generations, in comparison to the other three.

For the top 3 configurations we performed a Kruskal-Walli's test and failed to reject the null hypotheses ($p>0.01$), which means there are no significant differences between their fitness values. Given this, we decided to select configuration 0, which uses swap mutation (rate=1), group preserving order crossover (rate 0.8), rank selection ($\lambda = 10$) as our best GA configuration, as it has the highest median fitness and lower variance (Figure 25).

5.2 Simulated Annealing Results

When analysing aggregation of configurations, we observe that using scramble mutation as the random neighbour generator is significantly better than using swap mutation (Figure 27), contrary to what happened with the GA.

This is likely because for this algorithm the neighbour search is the only operation that introduces diversity in the solution to move towards the global optimum. In such cases, the higher the disruption, the faster the algorithm can move towards a better solution. However when we compare different aggregated scramble sizes (k) we see that the smaller the scramble size (k = 2, 3) the better the results with these configurations, and that higher scramble sizes (k = 1) present higher variability (bigger IQR) (Figure 28).

When aggregating by the control parameter (C) and by the decay rate (H) there were no significant differences for fitness distributions (Figure 29 and Figure 30), as these parameters don't seem to be the predominant determinants within this range.

For the top configurations they all perform similarly well achieving a good solution over 10,000 fitness evaluations. The exception was the configuration with scramble mutation with a large scramble size (k = 1) possibly since too much variability in neighbour generation leads to a less stable solution. We chose the configuration with the highest best median fitness value (Figure 31 and Figure 32) being the one with scramble mutation with small scramble size (k = 3) and a small decay rate (H = 1.01) for a high C value (accepting more worse solutions). This configuration was better over the other with statistical significance ($p < 0.05$) in the Dunn's test. (Figure 33).

5.3 Hill Climbing Results

Hill climbing was evaluated with 6 iterations to compare with the other algorithms for the same number of fitness evaluations (around 10,000), and with 100 iterations to see if the algorithm would converge to a better solution. Surprisingly, using our solution representation and with swap mutation as the neighbour function, the algorithm could explore the a wide local search space (1,792 neighbours) to eventually converge to a good solution after 30 iterations (53,760 fitness evaluations: $30 * 1792 = 53,760$), being more computationally expensive (Figure 34).

5.4 Comparison of the Algorithms

After selecting the best configuration for each algorithm, we compared the median best fitness evolution over 10,000 fitness evaluations to normalize for the algorithm complexity as a proxy for the runtime (Figure 35).

The results show that the GA and SA achieve almost double of the final median fitness (around 76,000) compared to the HC algorithm (39,400) within this fitness evaluation range. Among them, GA is the fastest to converge, reaching a high-quality solution in just 5,000 fitness evaluations, making it the most efficient for this optimization problem. This is likely due to the higher exploratory power of the GA driven by its combination crossover and mutation functions that introduce more high-quality diversity in the solution space.

The SA also converges to a good solution after 8,000 fitness evaluations benefiting from the diversity introduced by the scramble mutation operator. In contrast, HC, which uses only swap mutation, converges much more slowly.

The distribution of the best final fitness values after 10,000 fitness evaluations is similar for GA and SA (Figure 36) and they are not significantly different ($p = 0.622$) according to Mann-Whitney U test. However, if we compare the final best fitness values at 5,000 fitness evaluations (Figure 37), the GA is significantly better than SA ($p = 0.032$), which supports the previous conclusion that GA is the most efficient algorithm for this problem.

Conclusion

In this project, we investigated the performance of three optimization algorithms—Genetic Algorithm (GA), Simulated Annealing (SA), and Hill Climbing (HC)—for the Wedding Seating Problem, a combinatorial optimization task aimed at maximizing guest happiness based on pairwise relationships.

We designed a list-of-lists solution representation that preserves table structure and implemented custom crossover operators to minimize disruption within tables. Through extensive hyperparameter tuning, we found that for GA, the best-performing configuration used a low-disruption mutation operator (swap mutation) with a high mutation probability (1.0), enabling more effective exploration of the search space.

For SA, the optimal configuration used scramble mutation with a small scramble size (k = 3), which introduced enough variability to escape local optima without destabilizing the solution quality. In HC, using swap

mutation and a full neighbourhood exploration strategy (1,792 neighbours per iteration) allowed the algorithm to gradually improve and avoid becoming trapped in local optima, although at a higher computational cost.

Among all methods, GA demonstrated the fastest convergence and best final solution quality within the fitness evaluation budget, likely due to its population-based approach and the diversity introduced by crossover and mutation. These results underscore the importance of balancing solution structure preservation with controlled structural disruption to maintain diversity while navigating complex and wide search spaces.

References

Kruskal-Wallis Effect Size — kruskal_effsize • rstatix, https://rpkgs.datanovia.com/rstatix/reference/kruskal_effsize.html. Accessed 22 May 2025.

"Kruskal-Wallis-Test simply explained." *Datatab*, https://datatab.net/tutorial/kruskal-wallis-test.com Accessed 22 May 2025.

"Mann-Whitney U Test: Non-Parametric Hypothesis Testing." *Datatab*,https://datatab.net/tutorial/mannwhitney-u-test.com Accessed 22 May 2025.

Vanneschi, Leonardo, and Sara Silva. *Lectures on Intelligent Systems*. Springer International Publishing, 2023.

Appendix

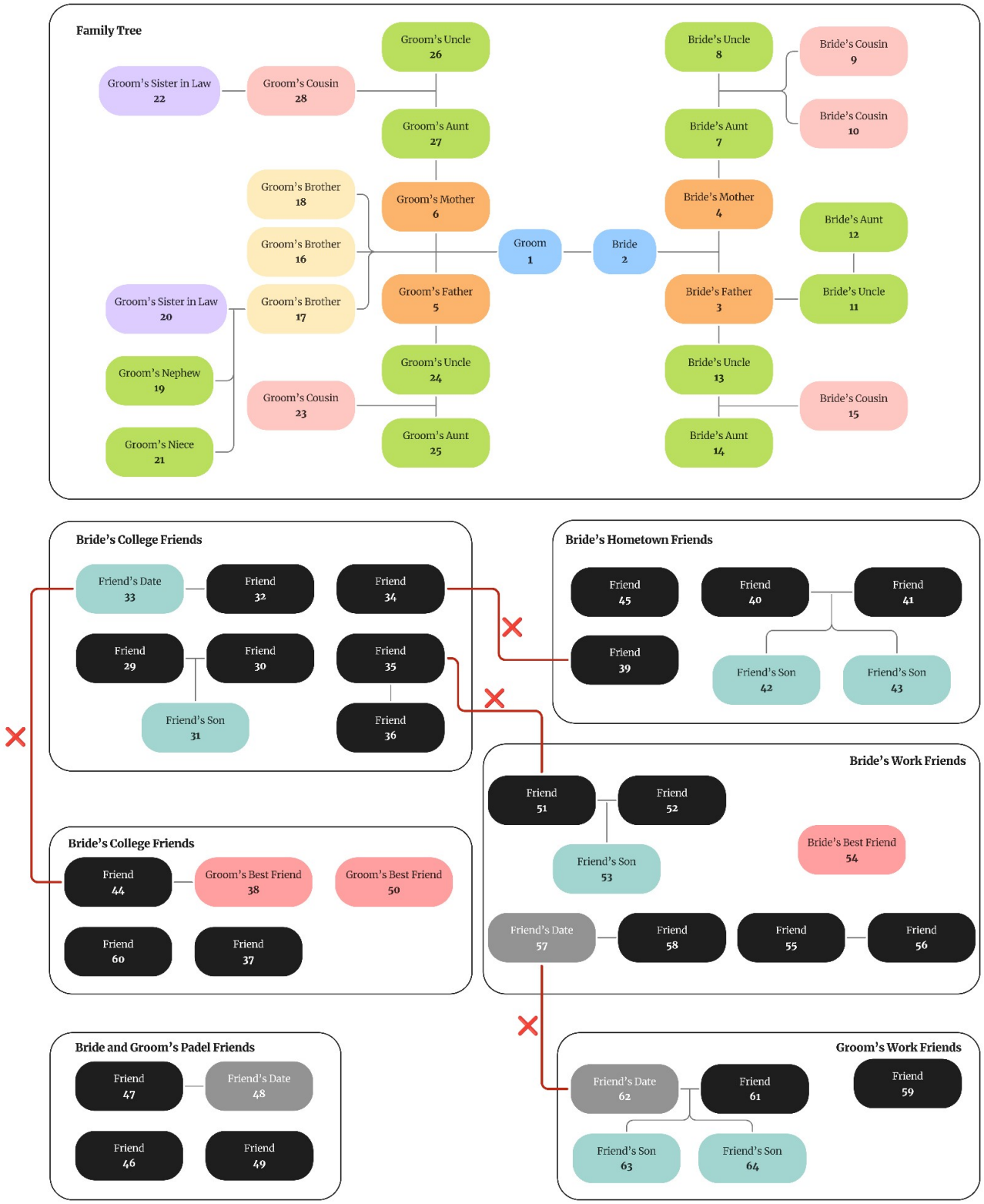


Figure 1: Guests' relationship chart

Relationship	**Value**
Bride or Groom	5000
Spouse or Date	2000
Best Friend	1000

Siblings	900
Parent or Child	700
Cousin	500
Aunt/Uncle or Niece/Nephew	300
Friend	0
Strangers	0
Enemies	-1000

Table 1: Assigned values for different guest relationships

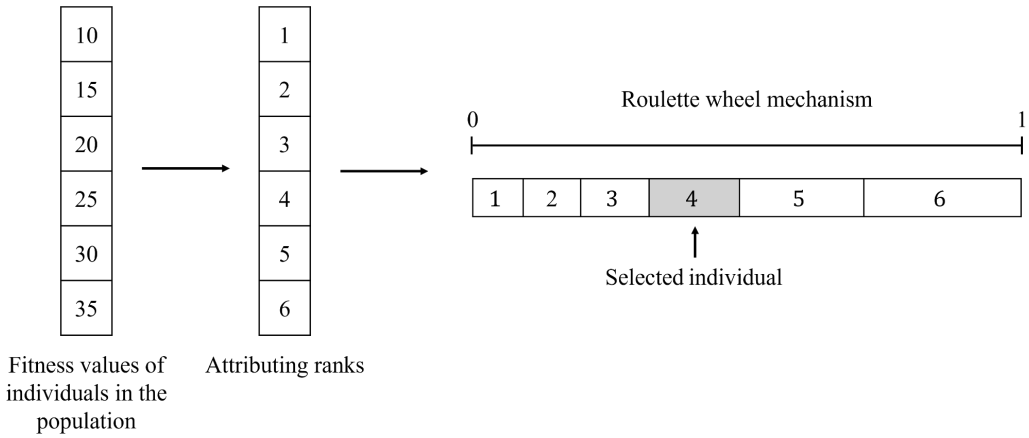


Figure 2: Example of rank selection for maximization problem

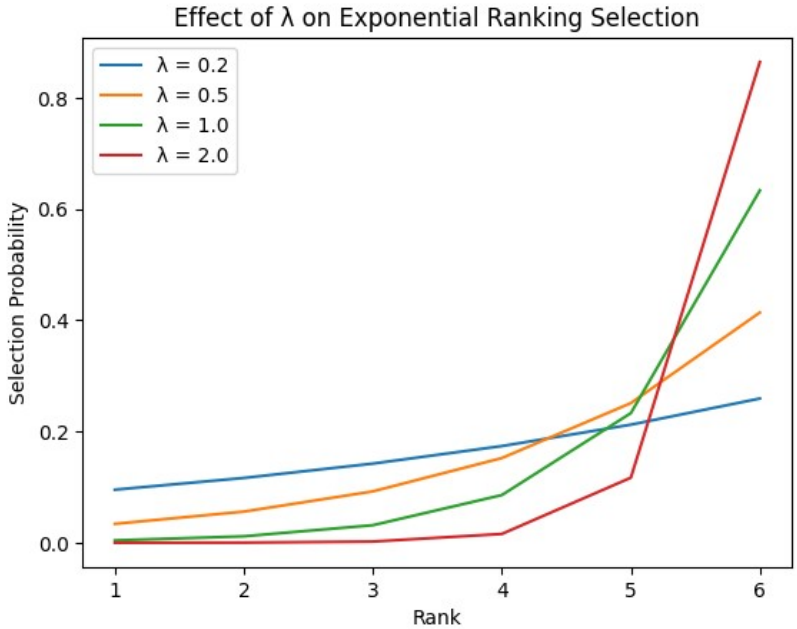


Figure 3: Effects of λ values in a maximization problem

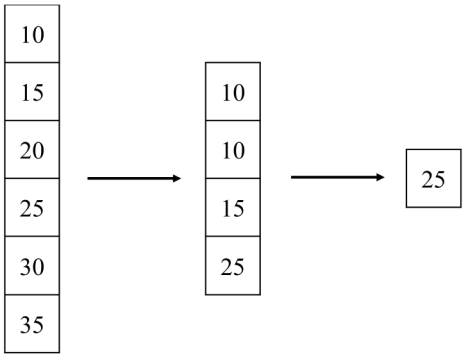
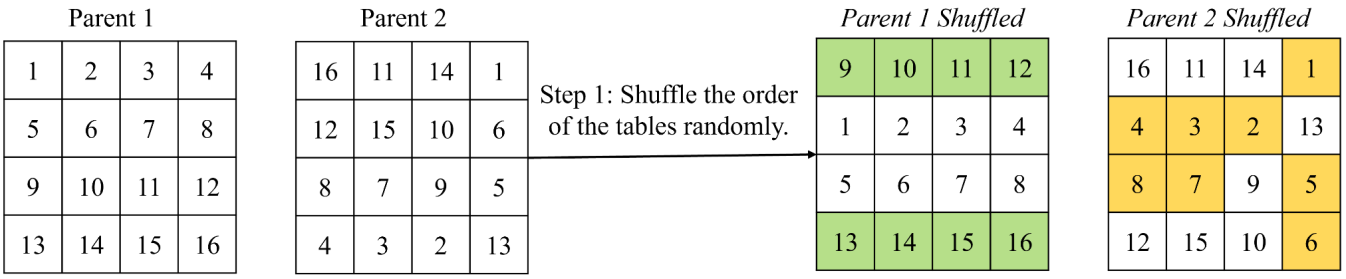


Figure 4: Example of tournament selection for maximization problem



Step 2: Select a random number of tables from *parent 1 shuffled* and plug them into **Child 1**: Table 1 and 4.

9	10	11	12
13	14	15	16

Step 3: Remaining positions are filled with parent 2 shuffled, in order. Skipping the already present guests.

9	10	11	12
13	14	15	16
1	4	3	2
8	7	5	6

Step 4: Shuffle the order of the tables. This will be **Child 1**.

13	14	15	16
1	4	3	2
9	10	11	12
8	7	5	6

Child 2: Follow the same procedure but preserve the order of the tables from parent 2.

Figure 5: Example of Order Preserving Crossover

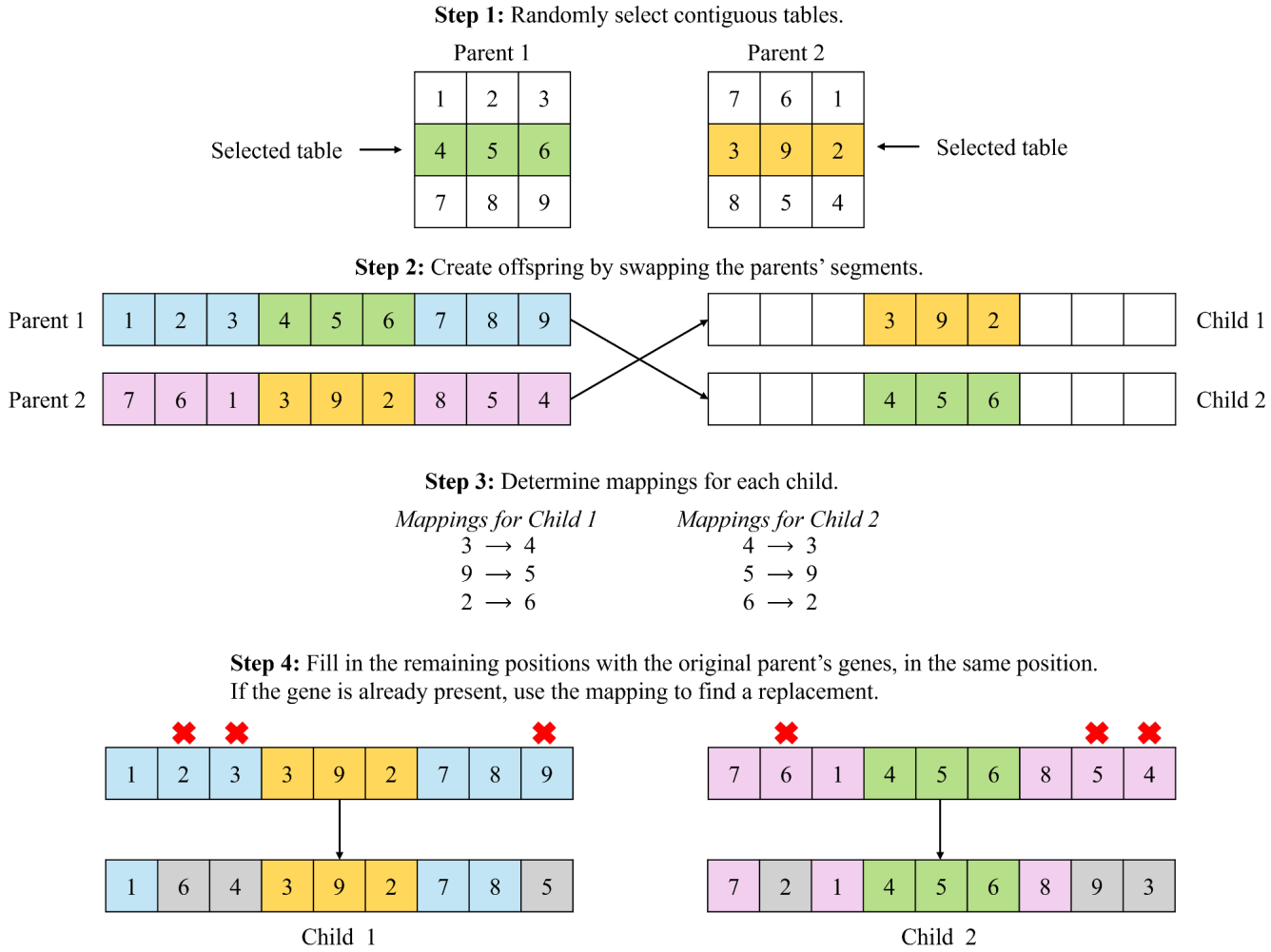


Figure 6: Example of Partially Mapped Crossover

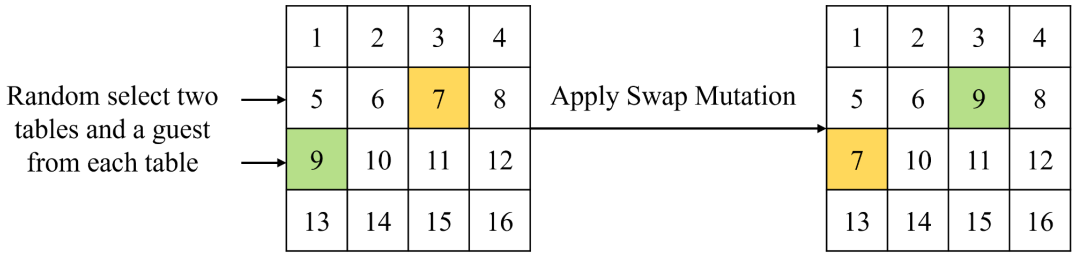


Figure 7: Example of Swap Mutation

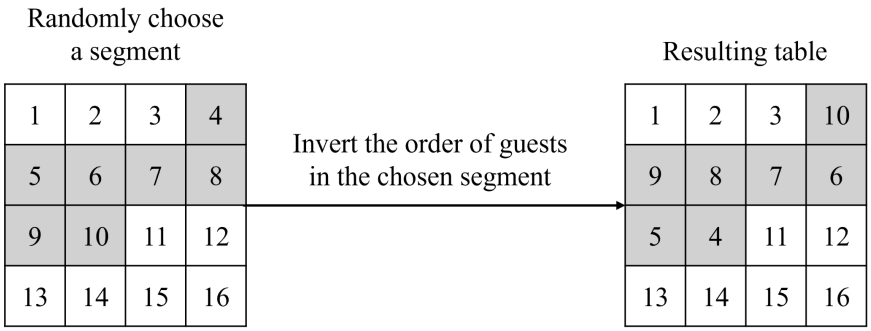


Figure 8: Example of Inversion Mutation

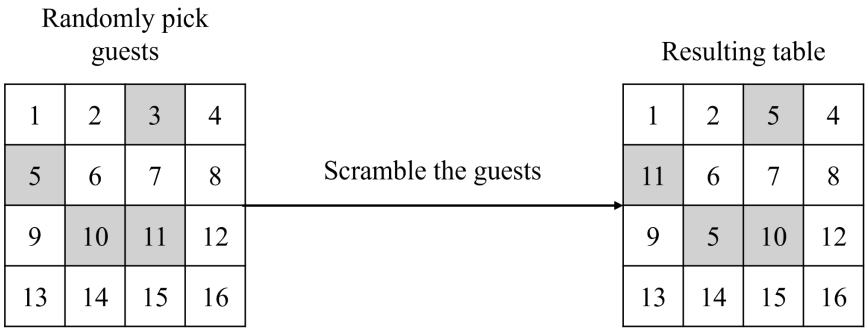


Figure 9: Example of Scramble Mutation

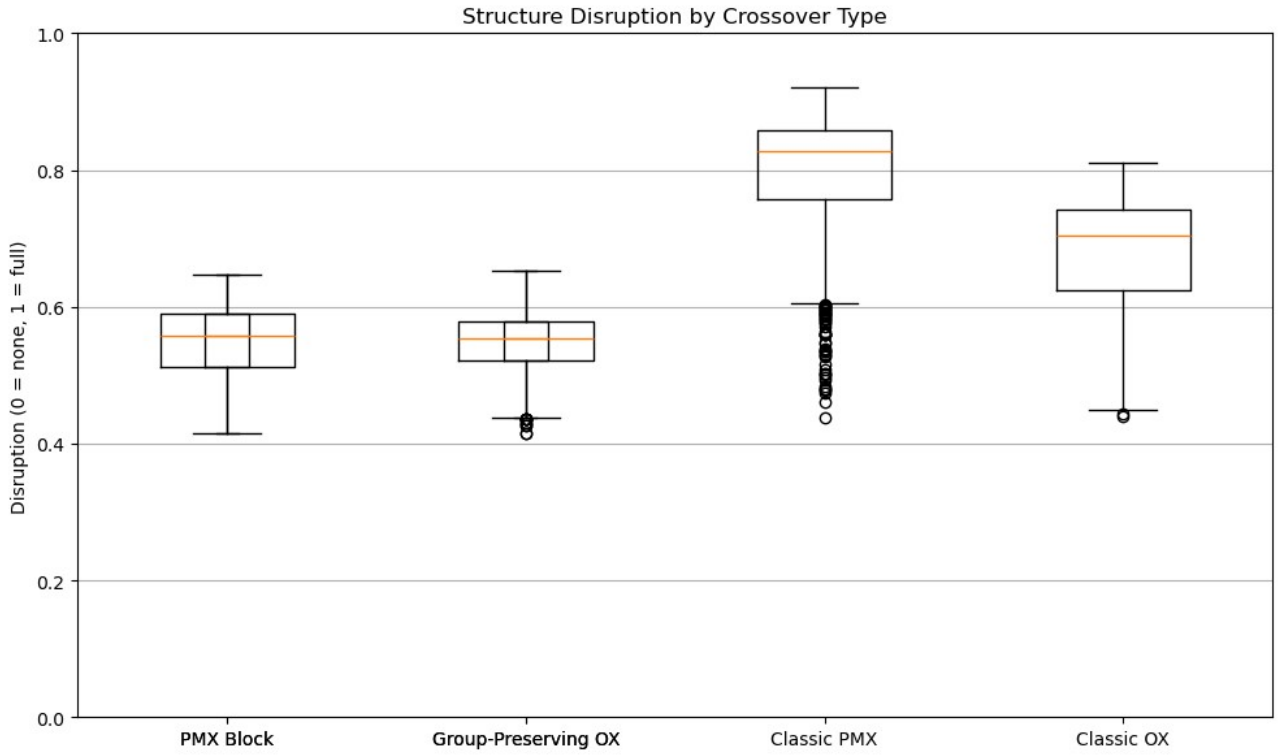


Figure 10: Structure Disruption with different crossover operators. Our custom crossovers (PMX block and Group preserving OX), and the classical PMX and OX. PMX - partially mapped crossover, OX - order crossover

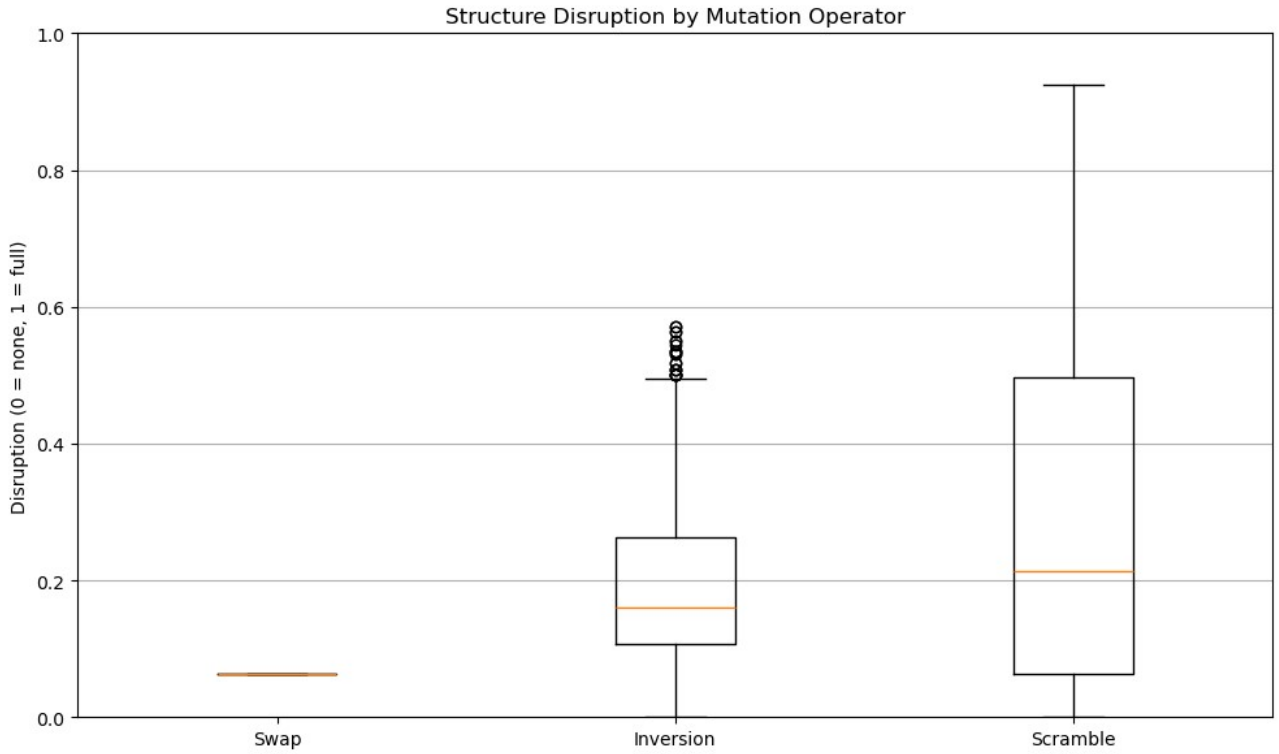


Figure 11: Structure disruption associated with different mutation operators

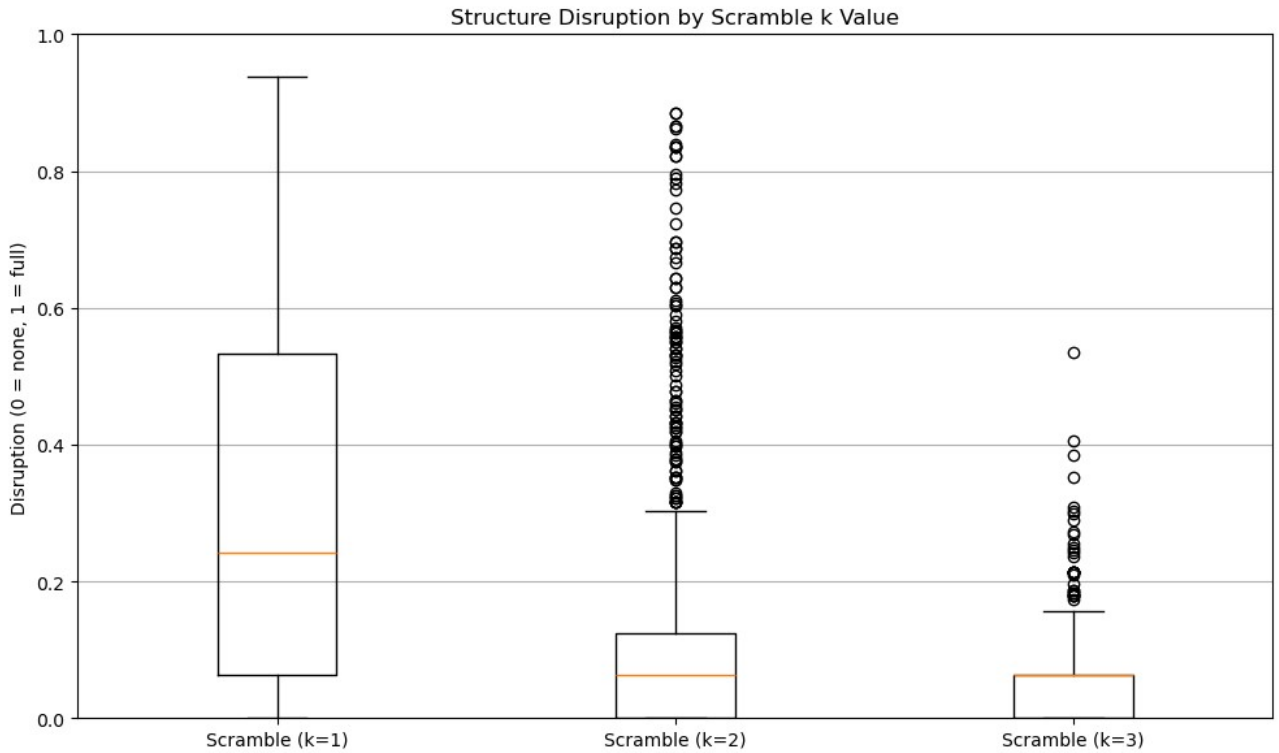


Figure 12: Structure disruption associated with different k values for the Scramble mutation

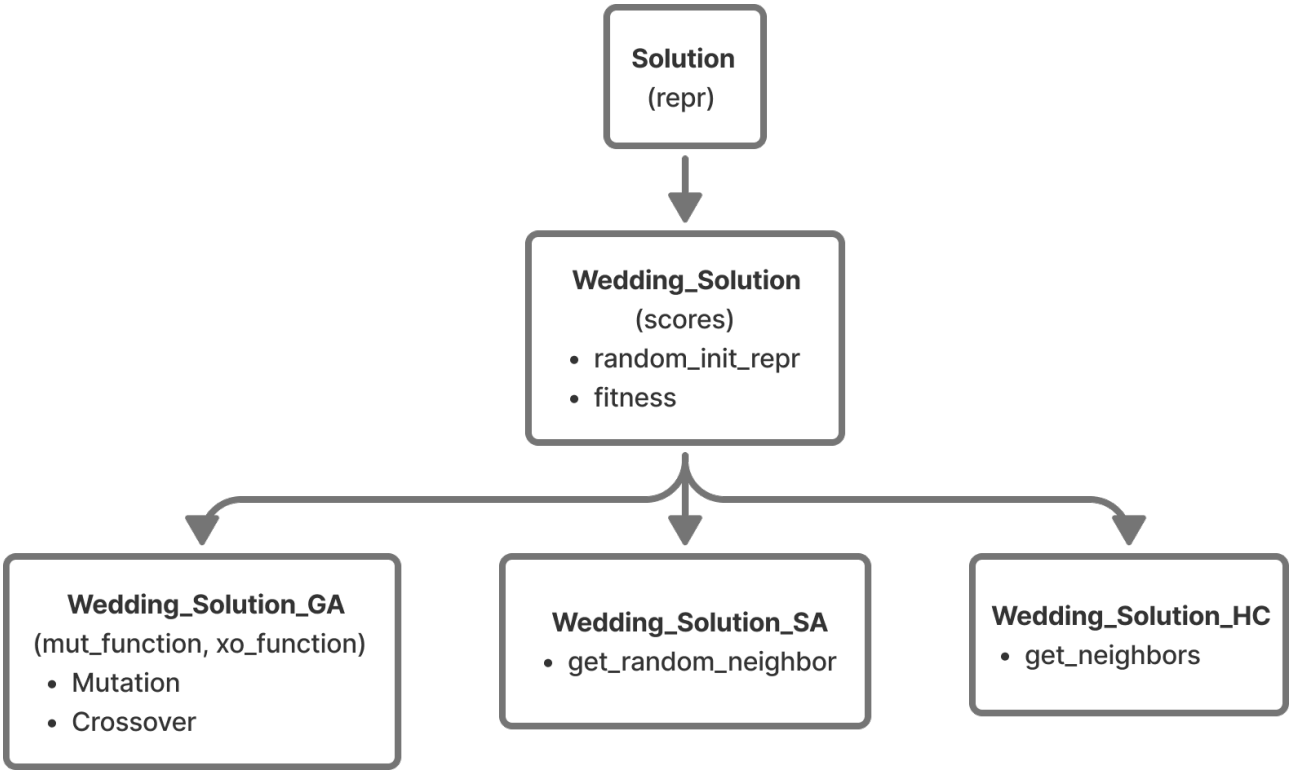


Figure 13: Classes and subclasses implementation for solutions.

Algorithm	**Hyperparameter group**	**Hyperparameter subgroup**	**Hyperparameters**
Genetic algorithm	Population size		50
	Max generations		100, 200
	Mutation operators		Swap mutation Scramble mutation Inversion mutation
	Crossover operators		PMX crossover Order crossover
	Selection operators		Rank selection Tournament selection
		Tournament size (tournament selection)	3, 5, 7, 10, 15
		Parameter λ (rank selection)	0.1, 1, 10

	Mutation probability		[0.1 - 1], step 0.1
	Crossover probability		[0.6 - 1], step 0.1
	Elitism		True, False
Simulated Annealing	Max iterations		100
	L		50, 100
	Neighbor operators		Swap mutation Scramble mutation
		Parameter k (scramble mutation)	1, 2, 3
	C		1, 10, 100, 200
	H		1.01, 1.1, 1.2, 1.3
Hill Climbing	Max iterations		6, 100

Table 2: Grid search hyperparameters for each algorithm

Algorithm	**Number of runs per configuration**	**Hyperparameters**	**Number of fitness evaluations**
Genetic algorithm	30	100 generations 50 population size	5,000
	100	200 generations 50 population size	10,000
Simulated annealing	30	100 iterations 50 neighbor search (L)	5,000
	100	100 iterations 100 neighbor search (L)	10,000
Hill Climbing	100	6 iterations	10,752
		100 iterations	179,200

Table 3: Number of fitness evaluations for each algorithm

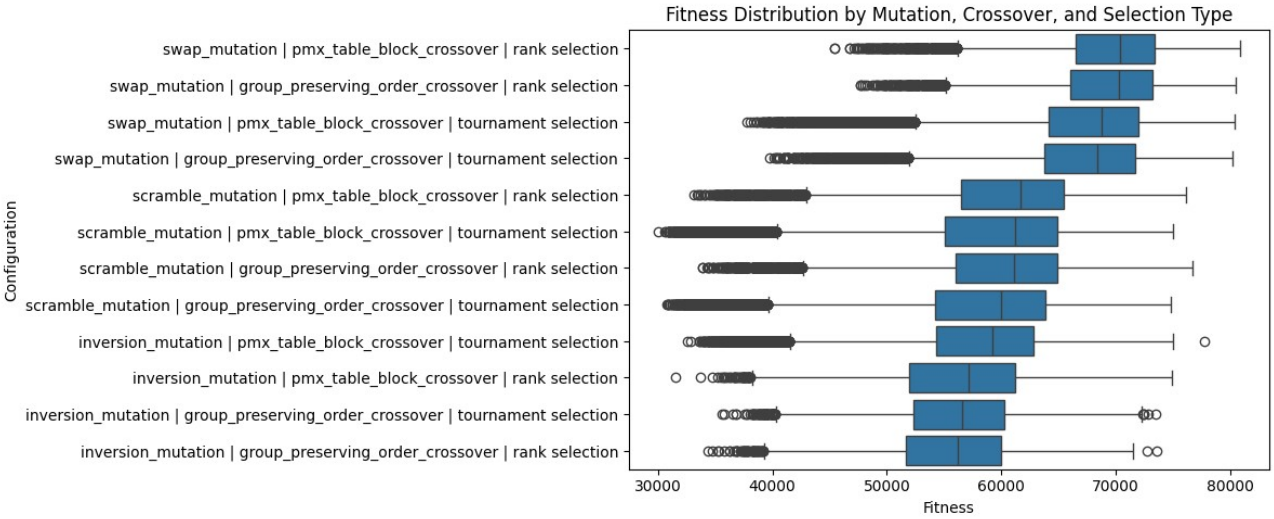


Figure 14: Boxplot of fitness values grouped by mutation, crossover and selection.

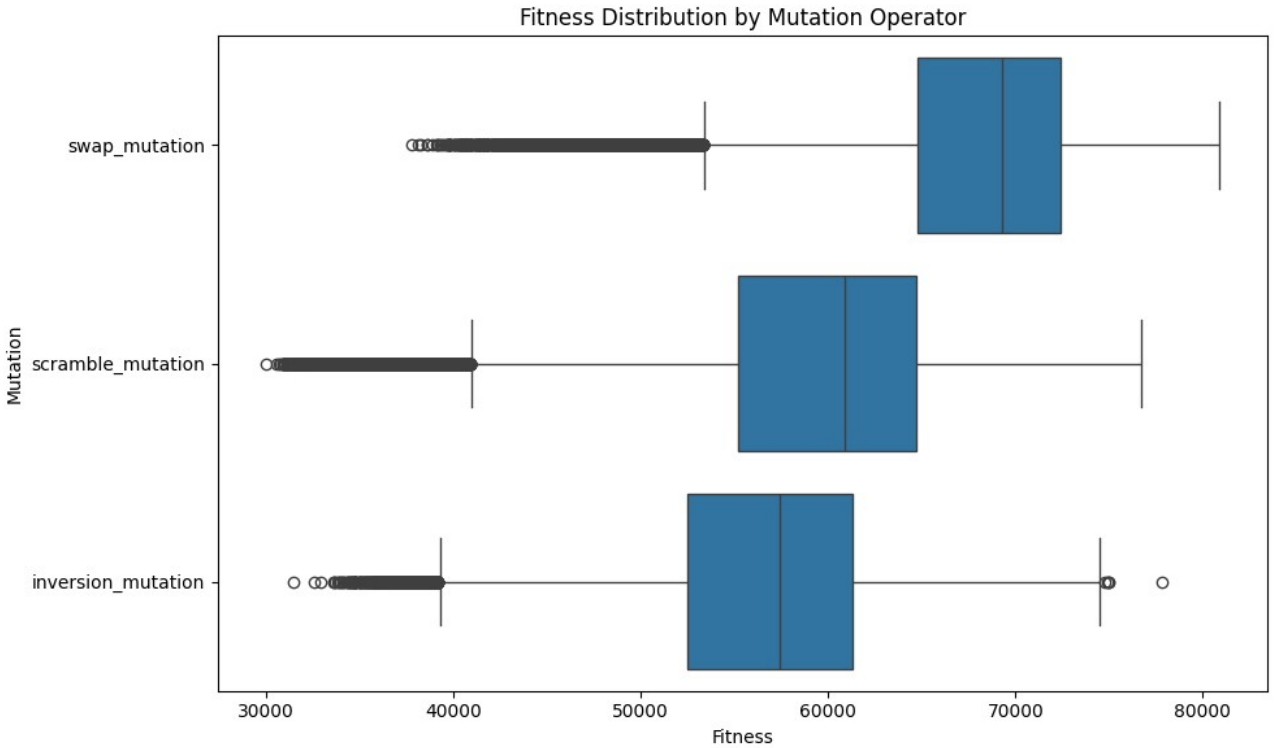


Figure 15: Boxplot of fitness values grouped by mutation operator.

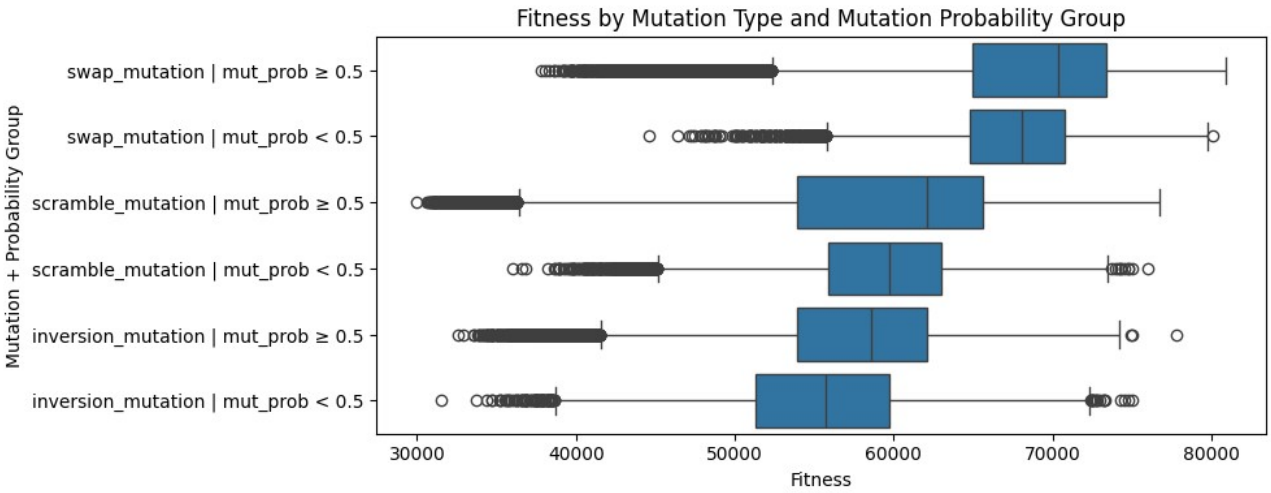


Figure 16: Boxplot of fitness values grouped by mutation operator and mutation probability.

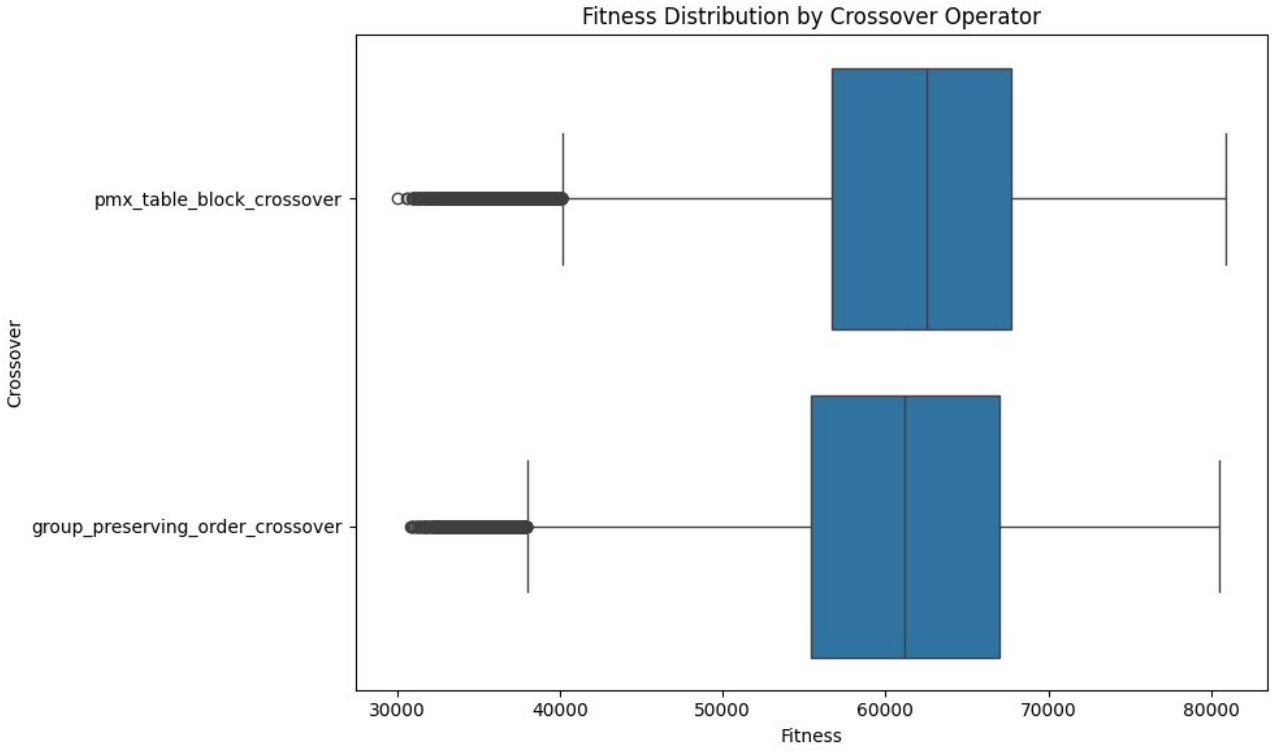


Figure 17: Boxplot of fitness values grouped by crossover operator.

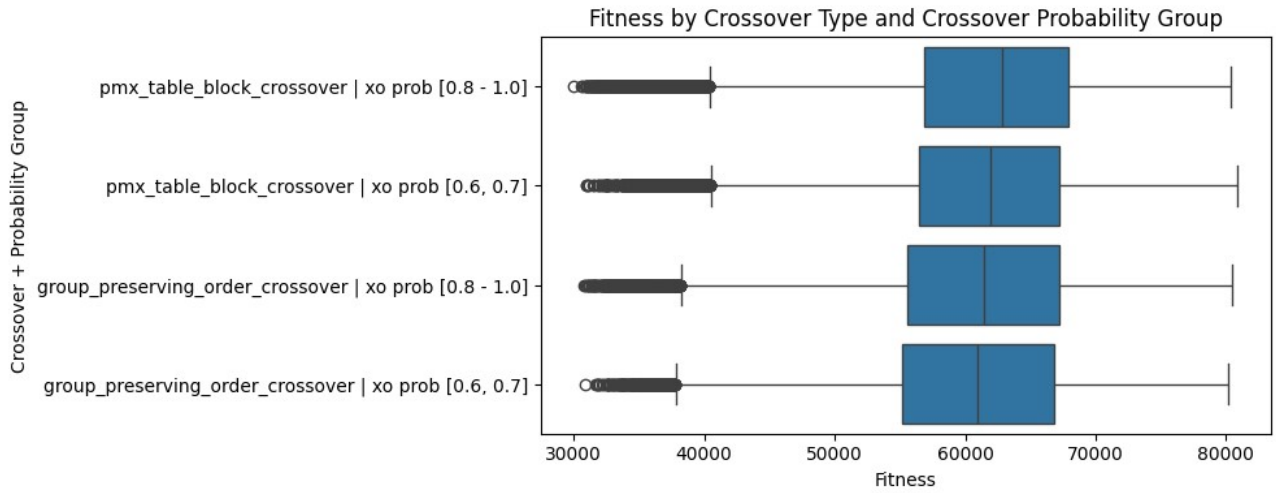


Figure 18: Boxplot of fitness values grouped by crossover operator and crossover probability.

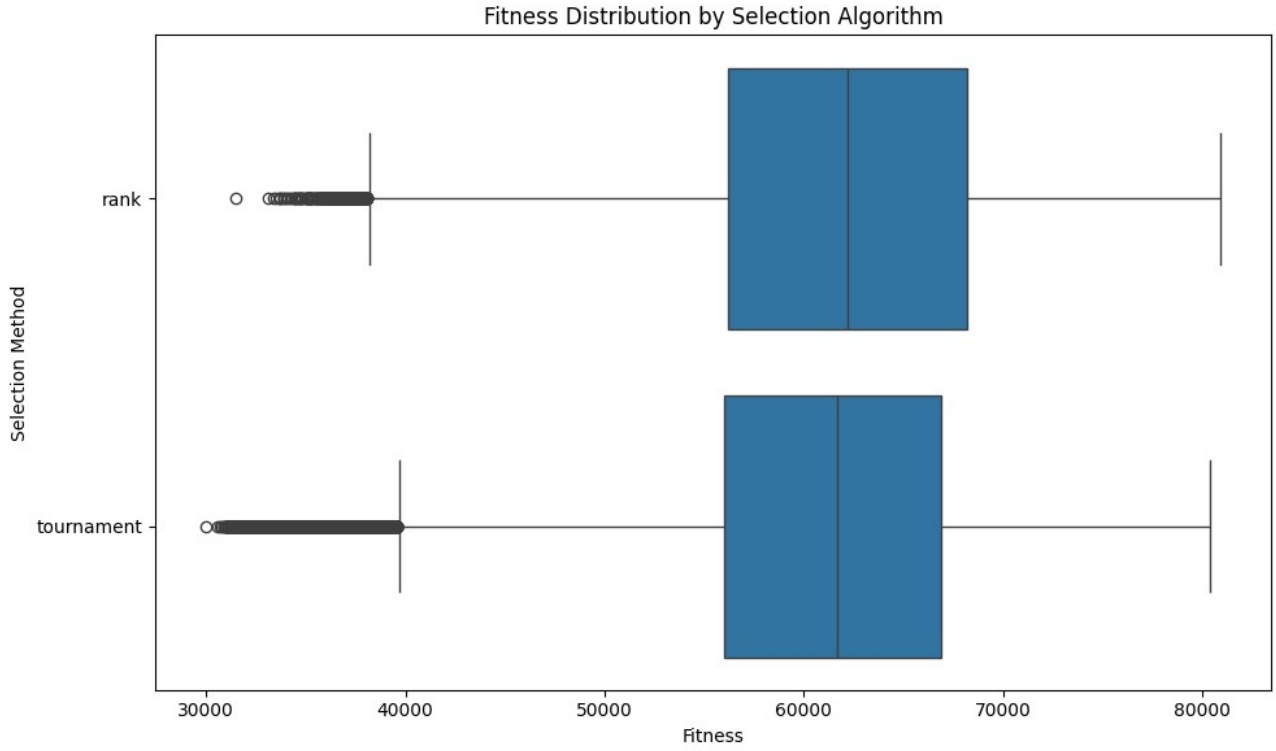


Figure 19: Boxplot of fitness values grouped by crossover algorithm.

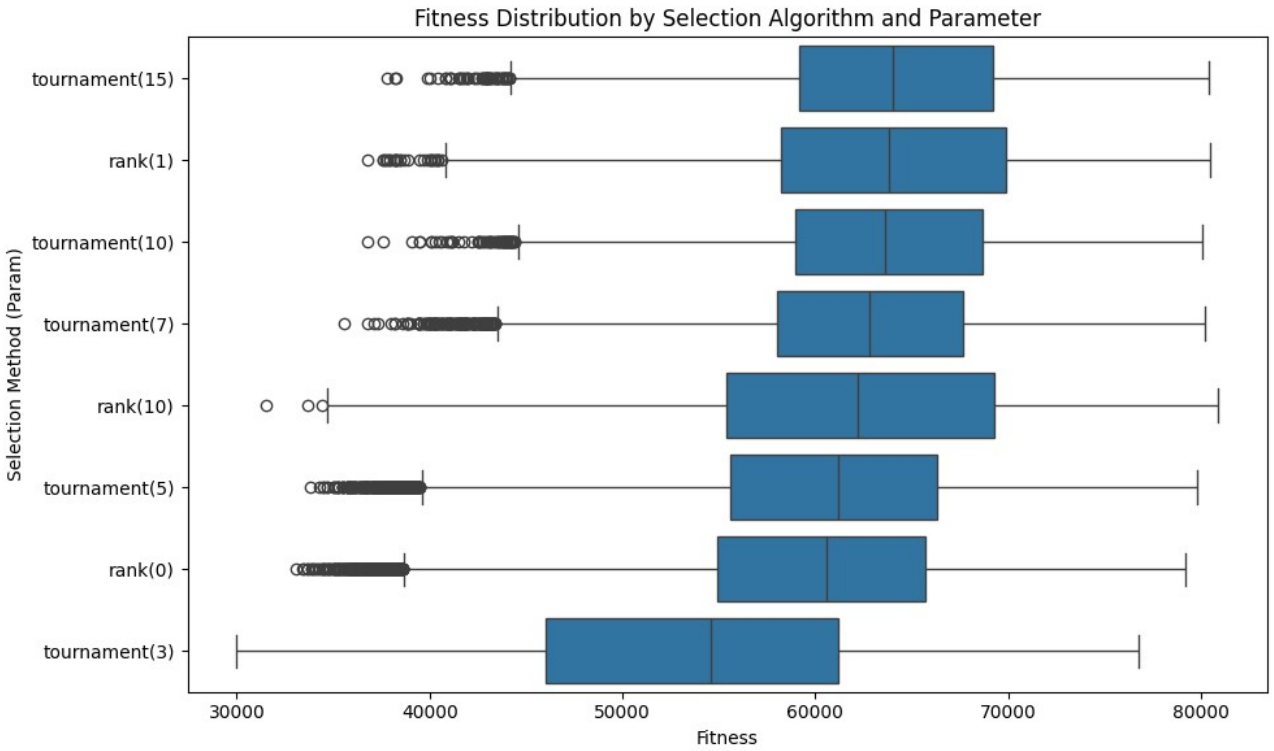


Figure 20: Boxplot of fitness values grouped by selection algorithm and parameters.

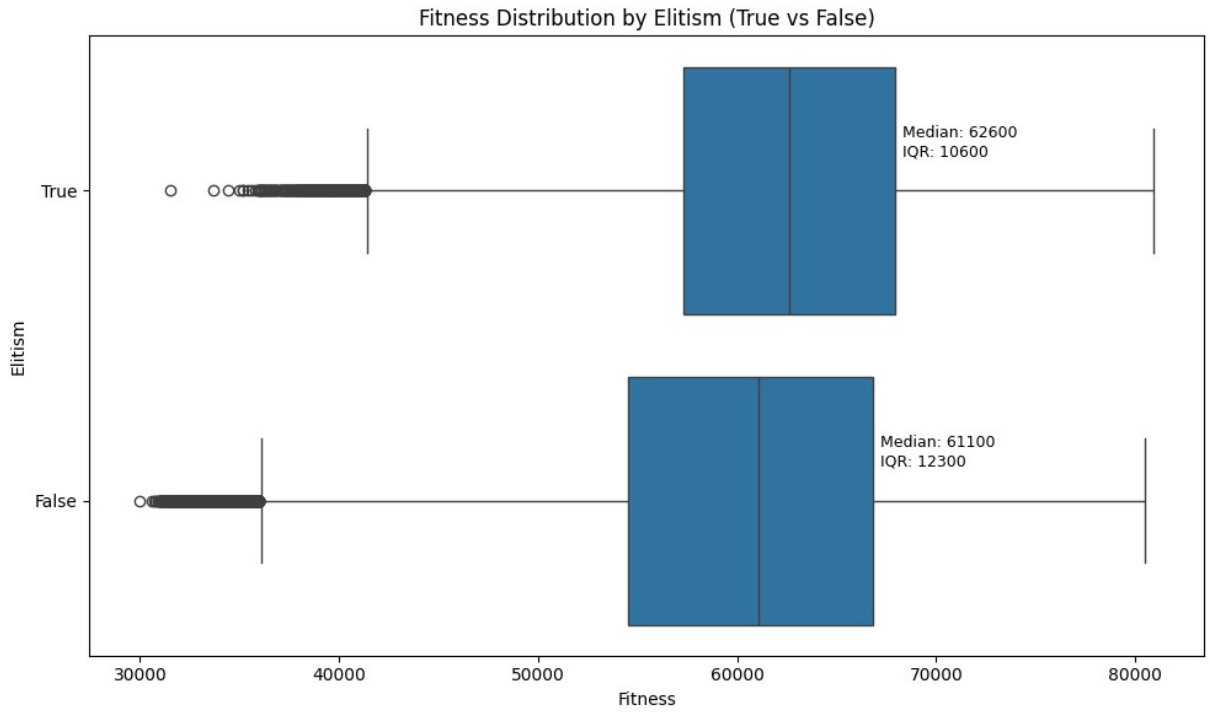


Figure 21: Boxplot aggregation by elitism

Number	Configuration
0	swap_mutation | group_preserving_order_crossover | rank(10.0) | xo=0.8 | mut=1.0 | elitism=True
1	swap_mutation | pmx_table_block_crossover | tournament(15.0) | xo=1.0 | mut=0.7 | elitism=False
2	swap_mutation | pmx_table_block_crossover | rank(1.0) | xo=0.9 | mut=1.0 | elitism=True

3	swap_mutation | pmx_table_block_crossover | rank(10.0) | xo=0.8 | mut=1.0 | elitism=True
4	swap_mutation | pmx_table_block_crossover | rank(1.0) | xo=0.7 | mut=0.9 | elitism=False
5	swap_mutation | pmx_table_block_crossover | rank(10.0) | xo=0.8 | mut=0.9 | elitism=True
6	swap_mutation | pmx_table_block_crossover | rank(1.0) | xo=0.8 | mut=0.8 | elitism=False
7	swap_mutation | pmx_table_block_crossover | rank(10.0) | xo=0.7 | mut=1.0 | elitism=True
8	swap_mutation | pmx_table_block_crossover | rank(10.0) | xo=0.6 | mut=1.0 | elitism=True
9	swap_mutation | pmx_table_block_crossover | rank(1.0) | xo=0.9 | mut=0.9 | elitism=True
10	swap_mutation | pmx_table_block_crossover | rank(10.0) | xo=0.6 | mut=0.7 | elitism=False
11	swap_mutation | pmx_table_block_crossover | rank(10.0) | xo=0.6 | mut=1.0 | elitism=False
12	scramble_mutation | group_preserving_order_crossover | rank(10.0) | xo=0.6 | mut=1.0 | elitism=True
13	inversion_mutation | pmx_table_block_crossover | tournament(10.0) | xo=1.0 | mut=0.7 | elitism=False

Tabela 4: Top 14 configurations numbered for reference

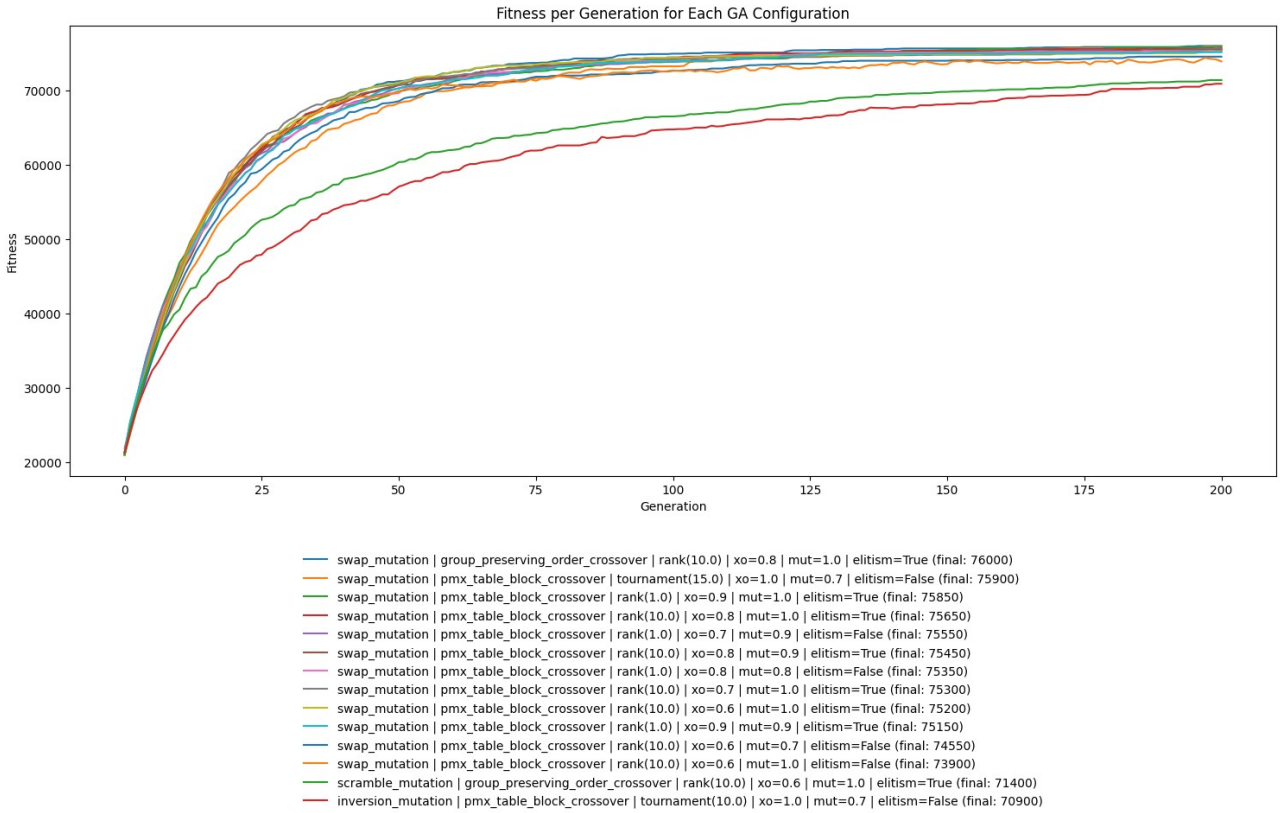


Figure 22: Fitness per generation for each GA configuration

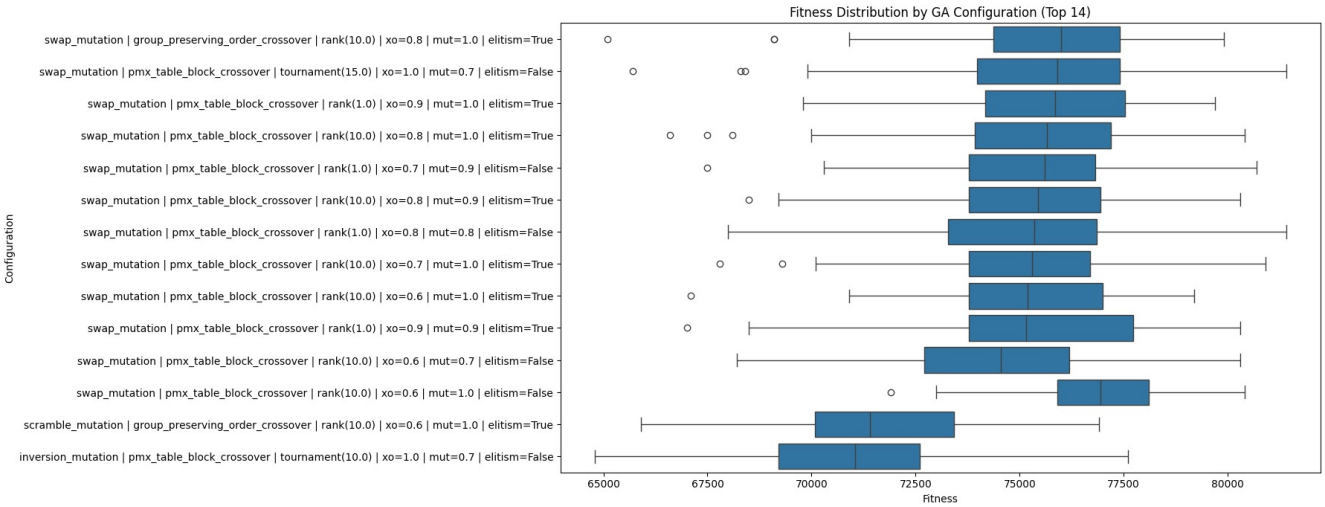


Figure 23: Boxplots of top 14 configurations fitness values

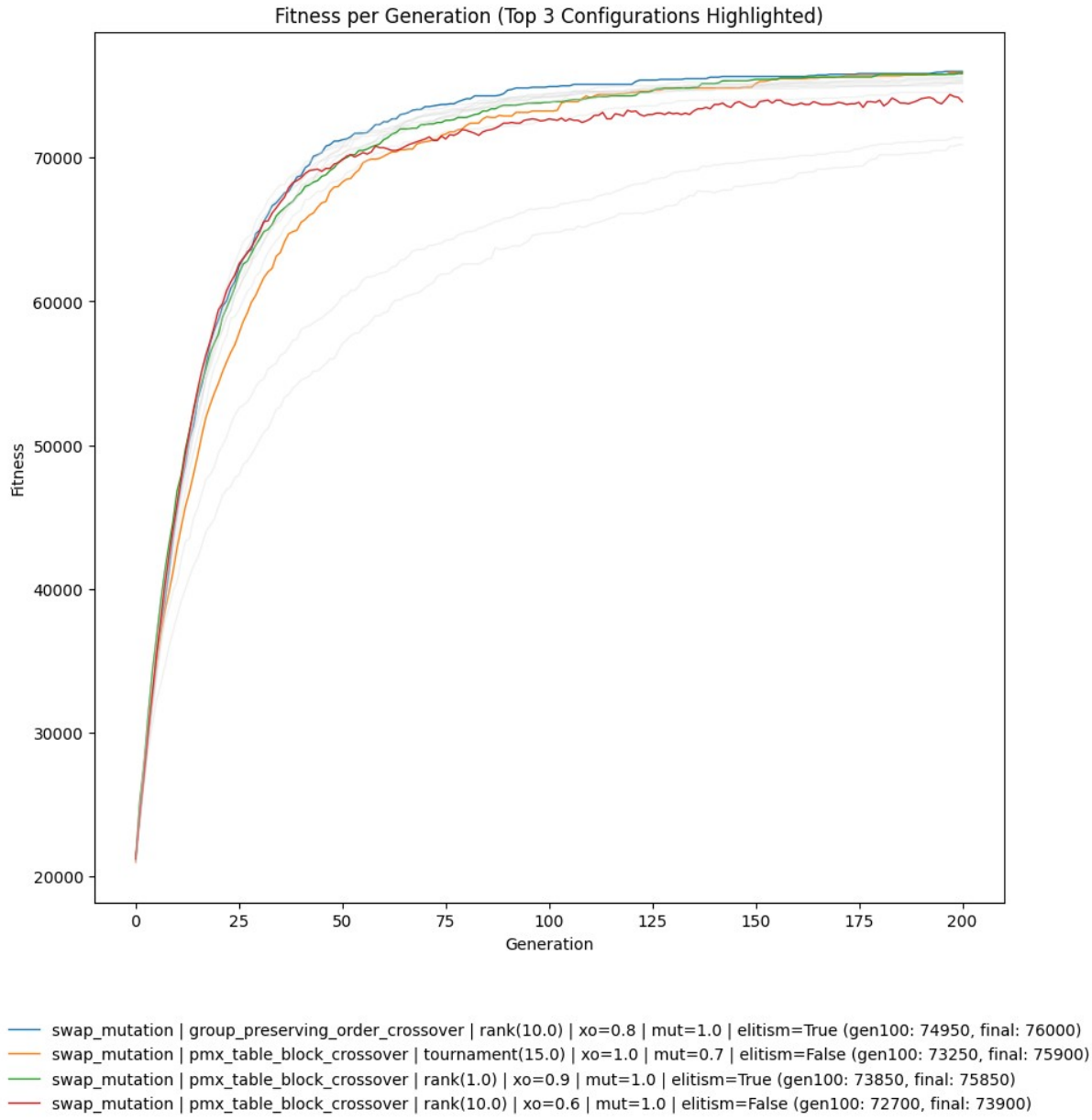


Figure 24: top 3 fitness per generation

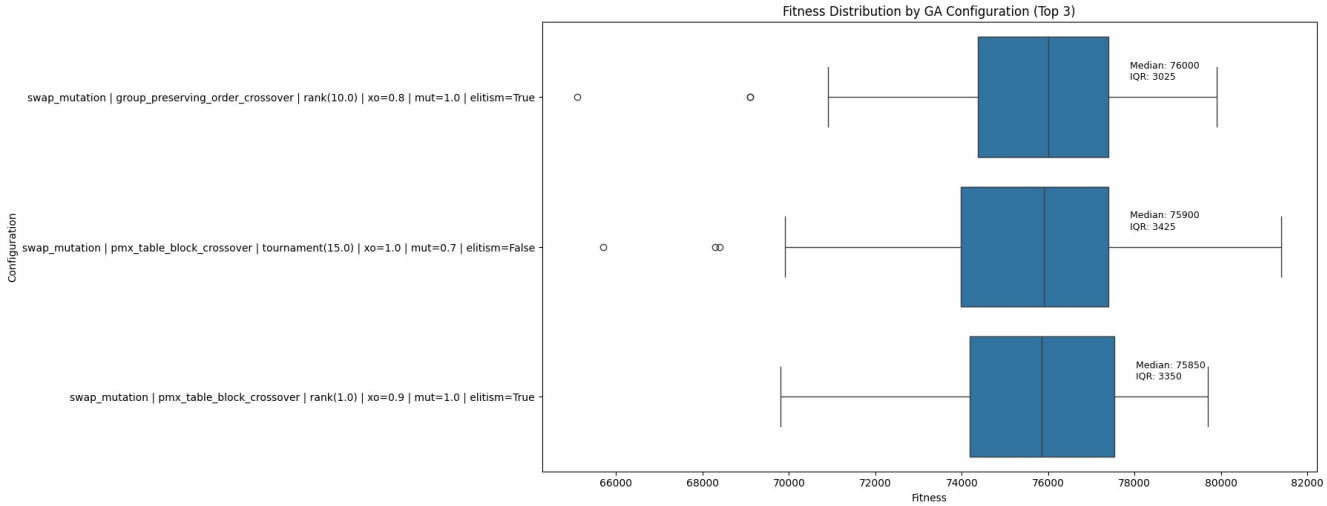


Figure 25: top 3 boxplot

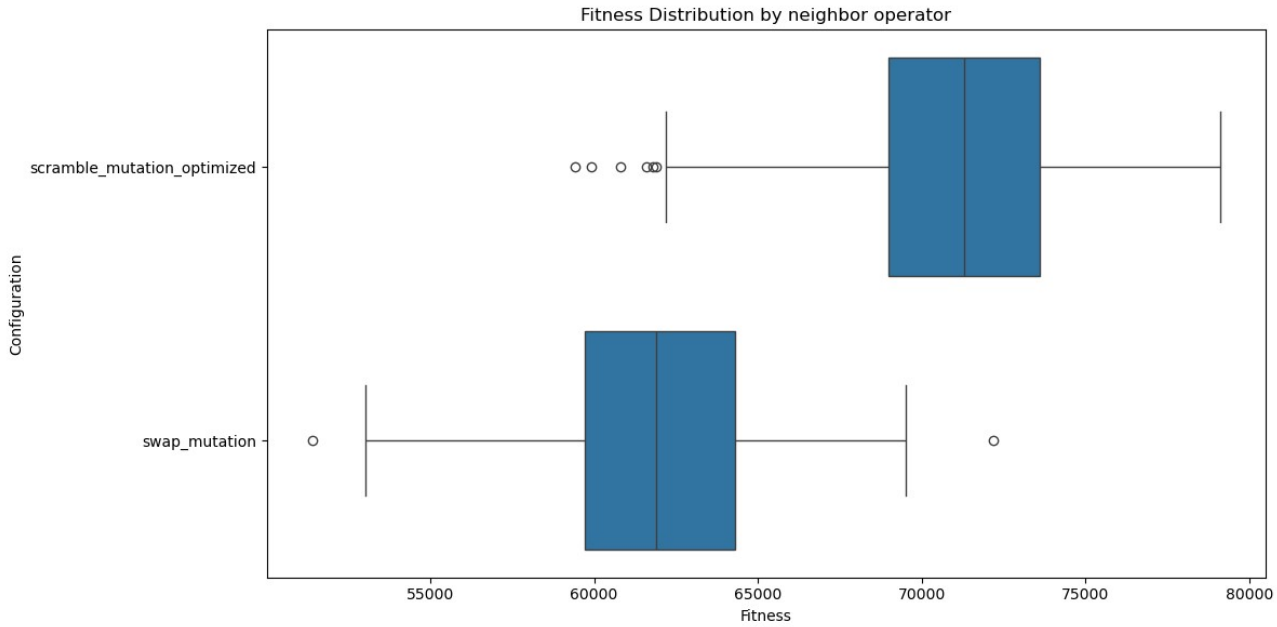


Figure 27: Fitness distribution by mutation operator (random neighbor function) for Simulated Annealing

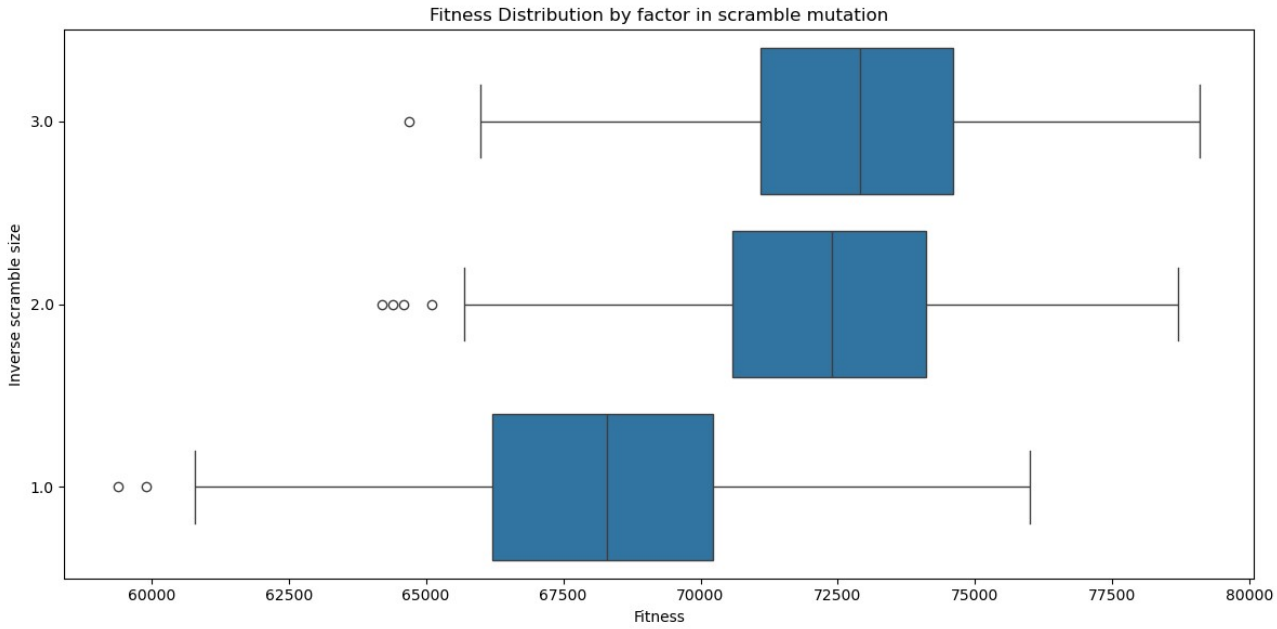


Figure 28: Fitness distribution by scramble size operator (k) for Simulated Annealing

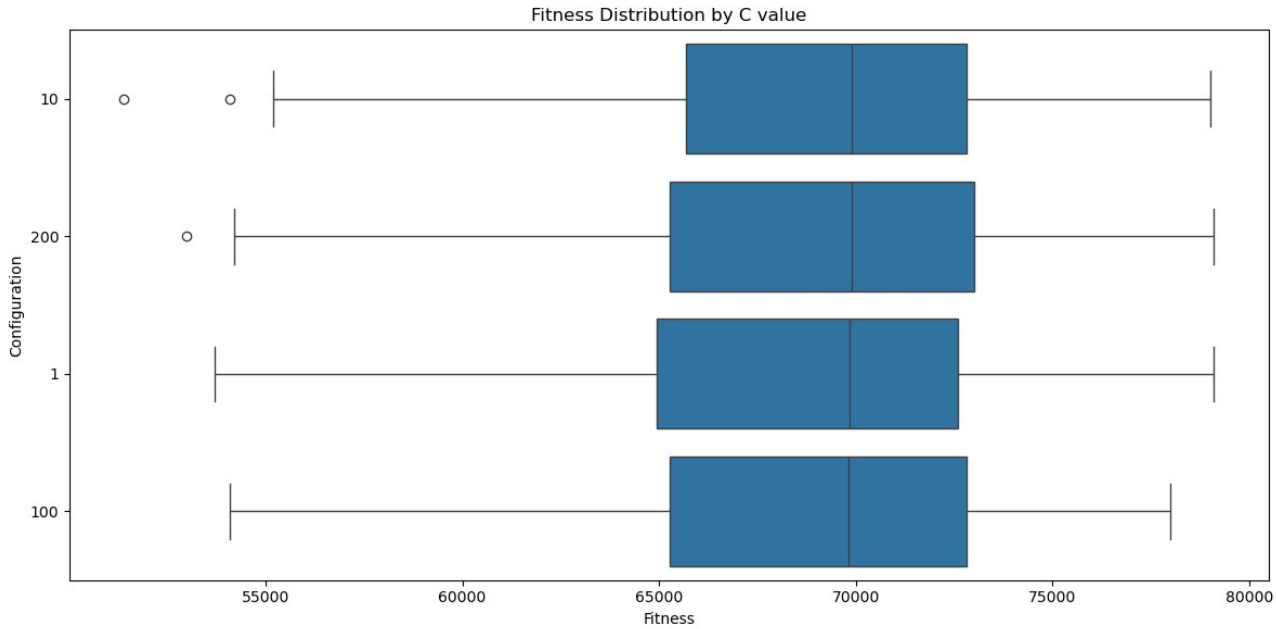


Figure 29: Fitness Distribution by C value for Simulated Annealing

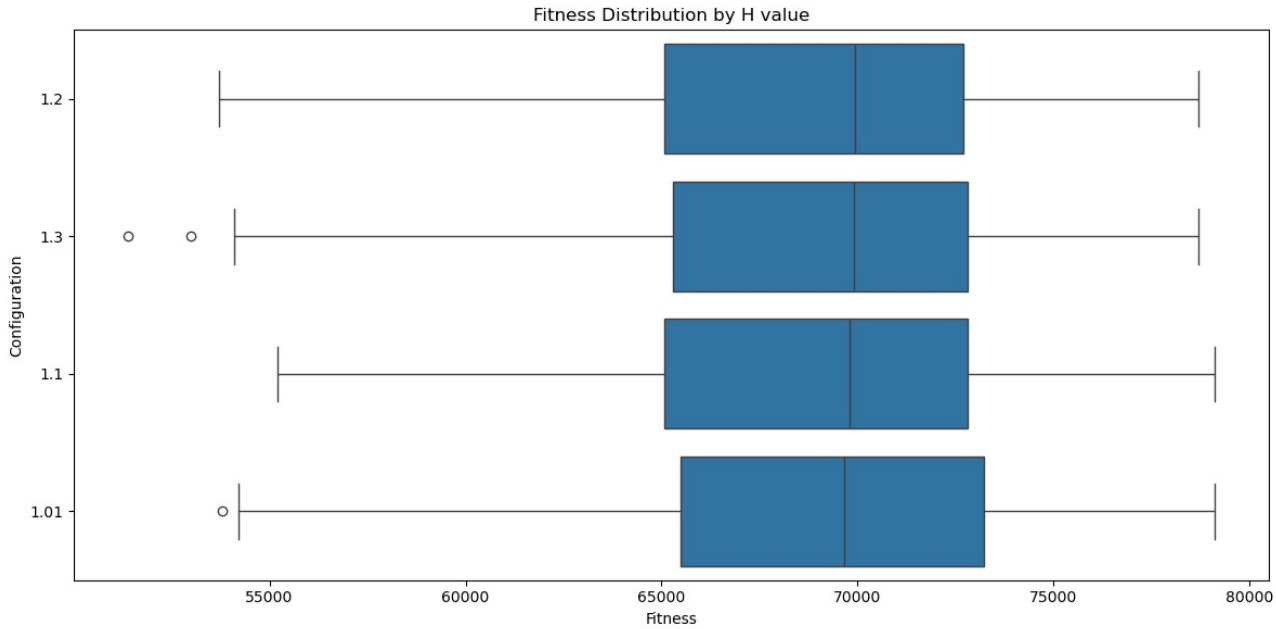


Figure 30: Fitness Distribution by H value for Simulated Annealing

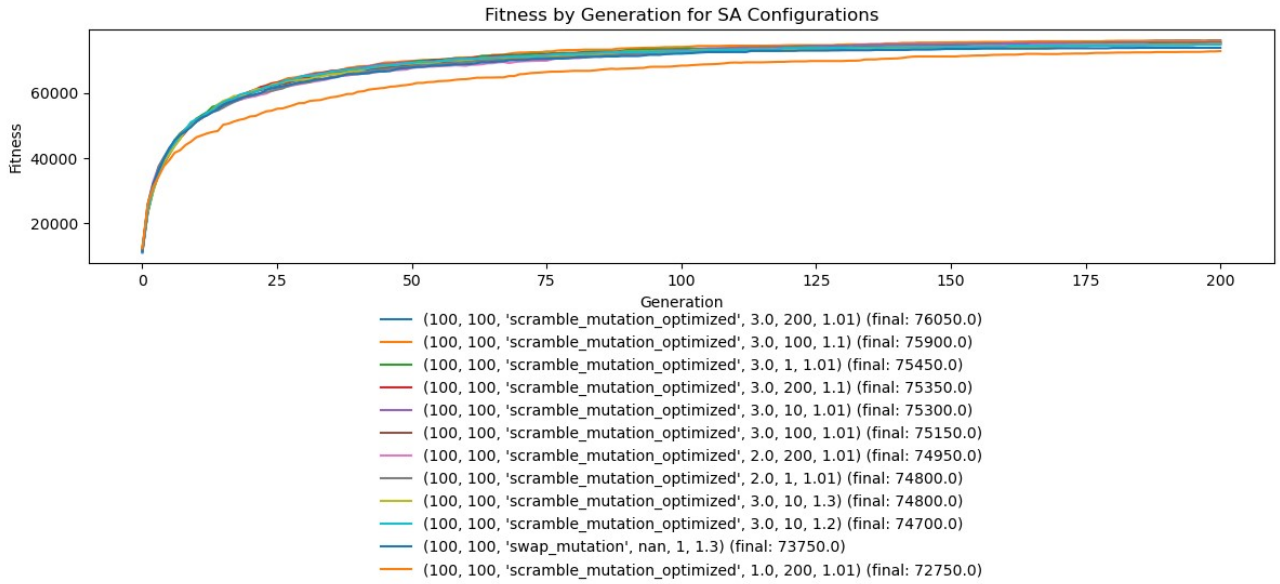


Figure 31: Median Fitness evolution for the top performing Simulated Annealing configurations

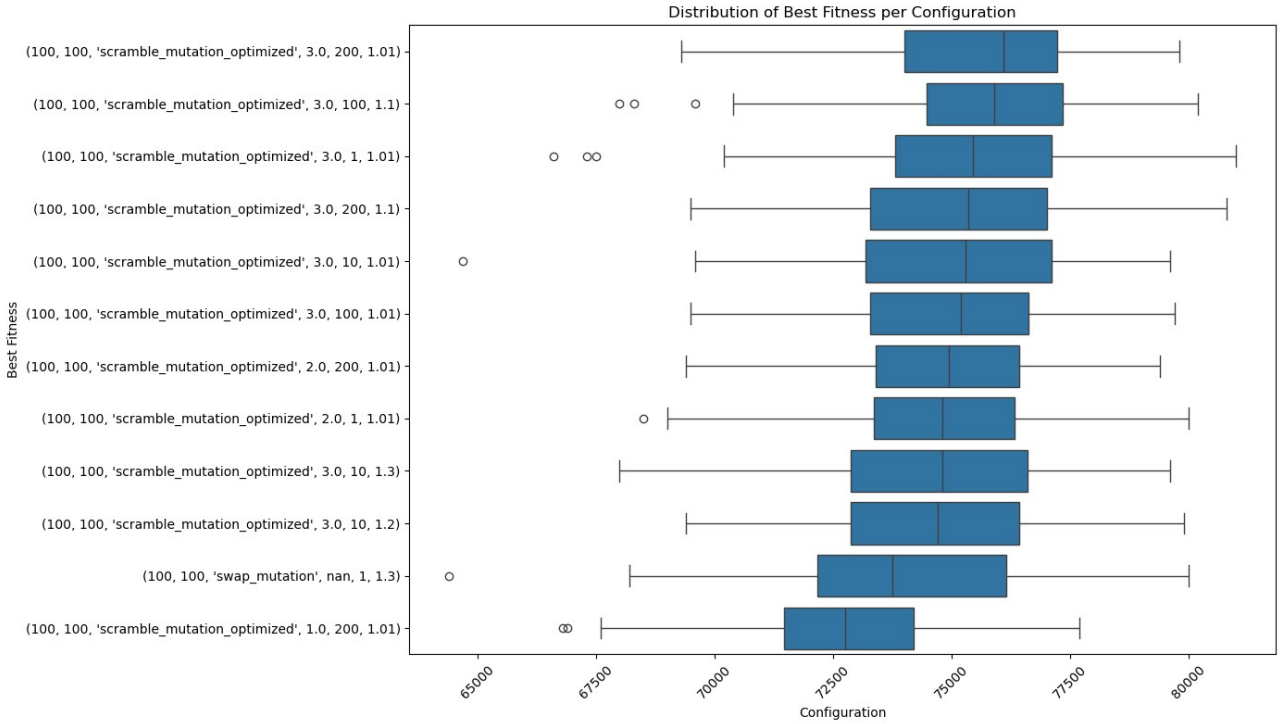


Figure 32: Best fitness distribution for the top performing Simulated Annealing configurations

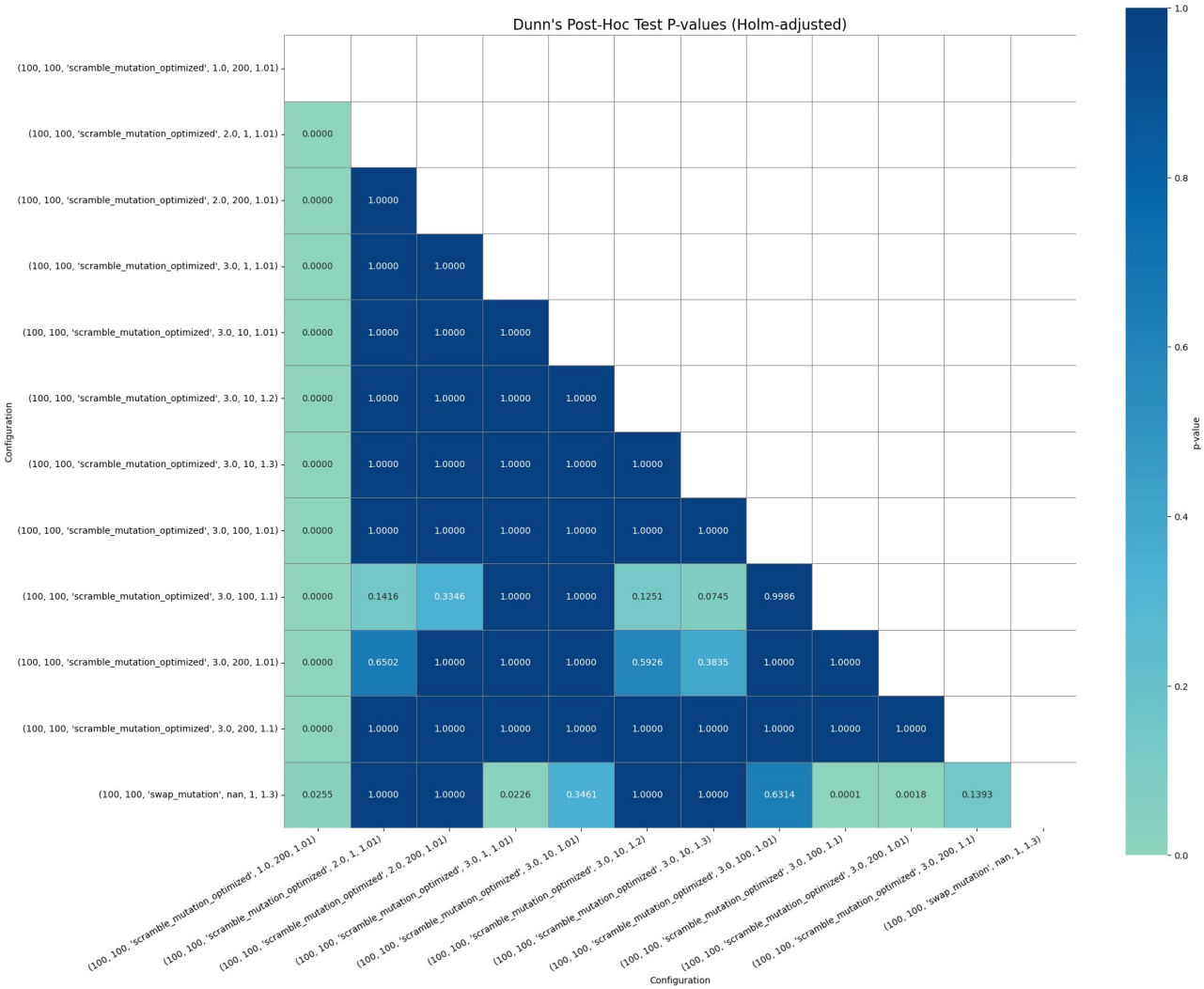


Figure 33: Dunn's Post-Hoc p-values for the top performing Simulated Annealing configurations

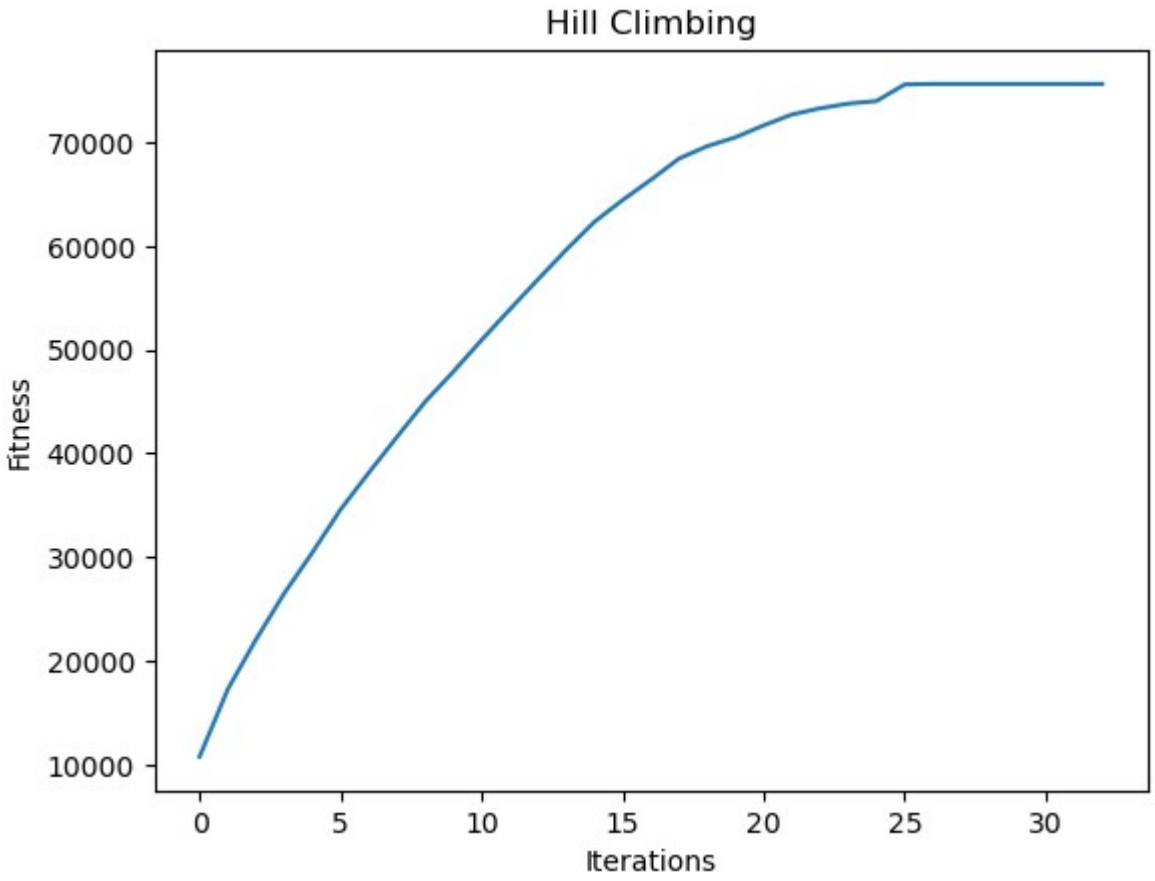


Figure 34: Median Best Fitness across iterations for Hill Climbing

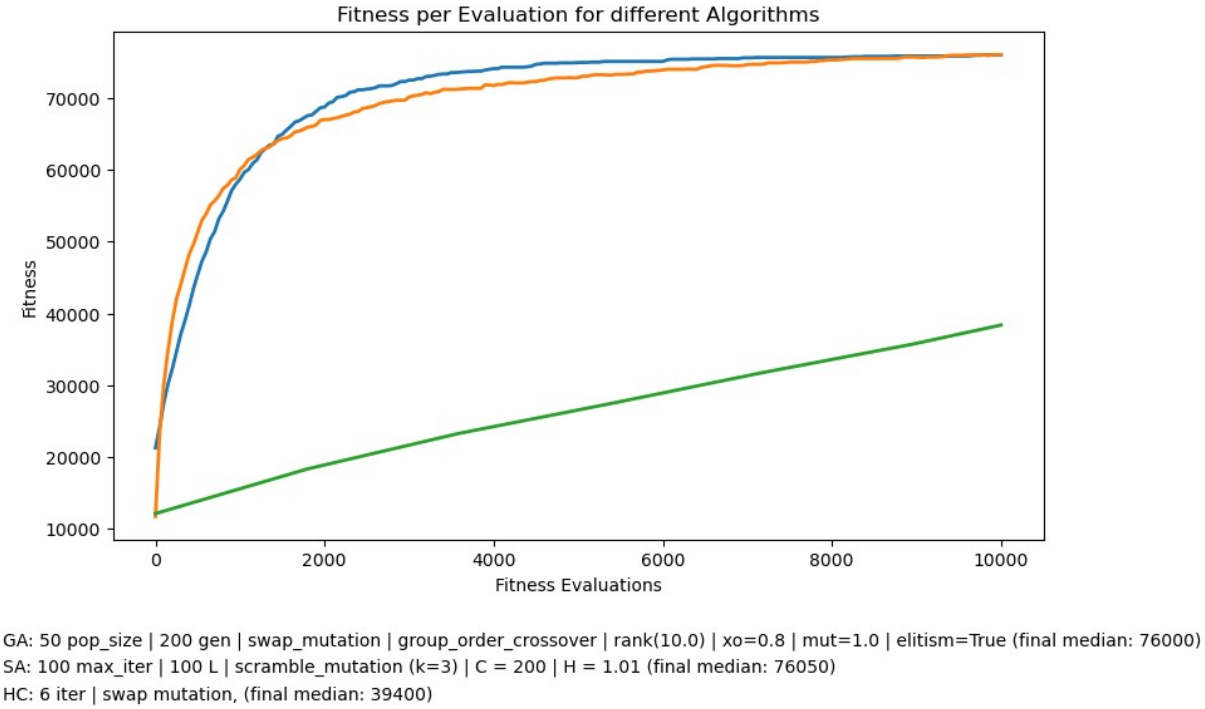


Figure 35: Median Best Fitness evolution over number of fitness evaluations for Genetic Algorithm (GA), Simulated Annealing (SA) and Hill Climbing (HC)

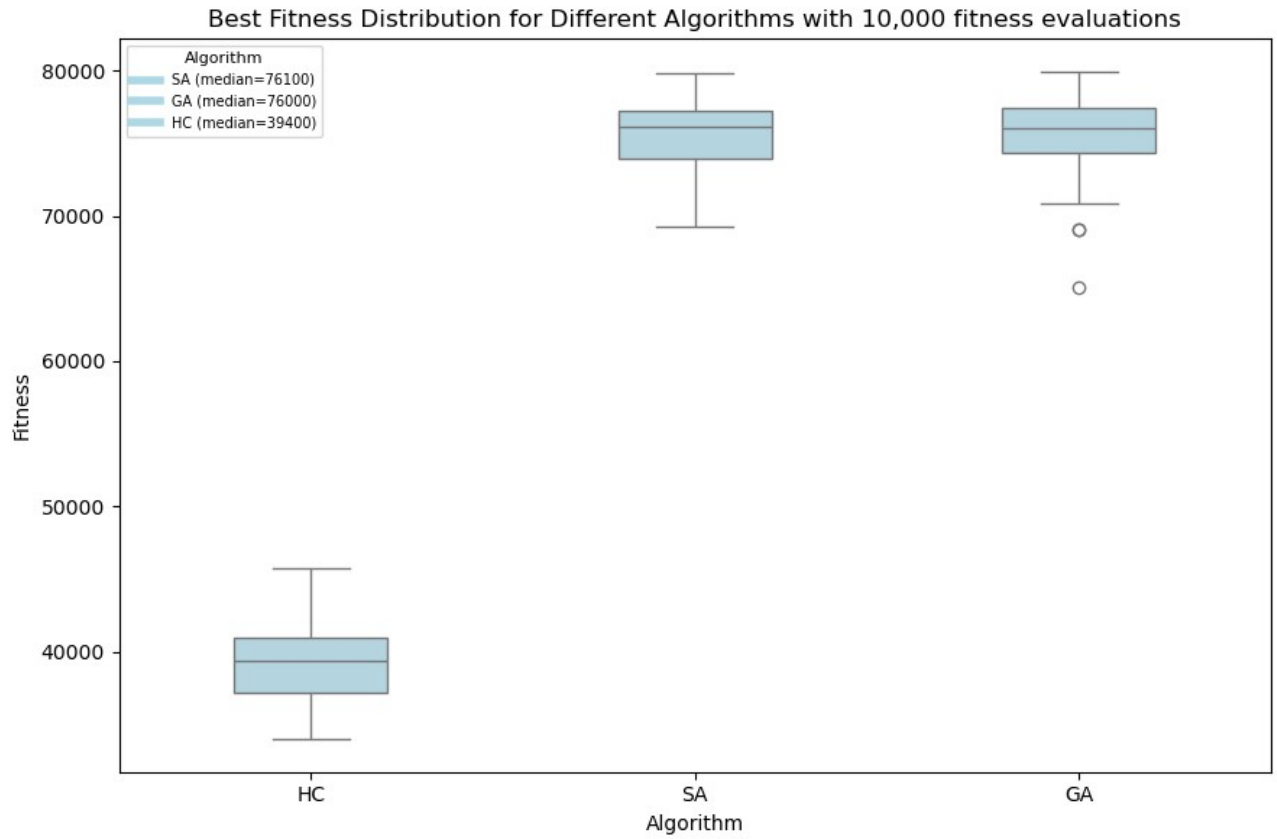


Figure 36: Best final fitness distribution after 10,00 fitness evaluations for Genetic Algorithm (GA), Simulated Annealing (SA) and Hill Climbing (HC)

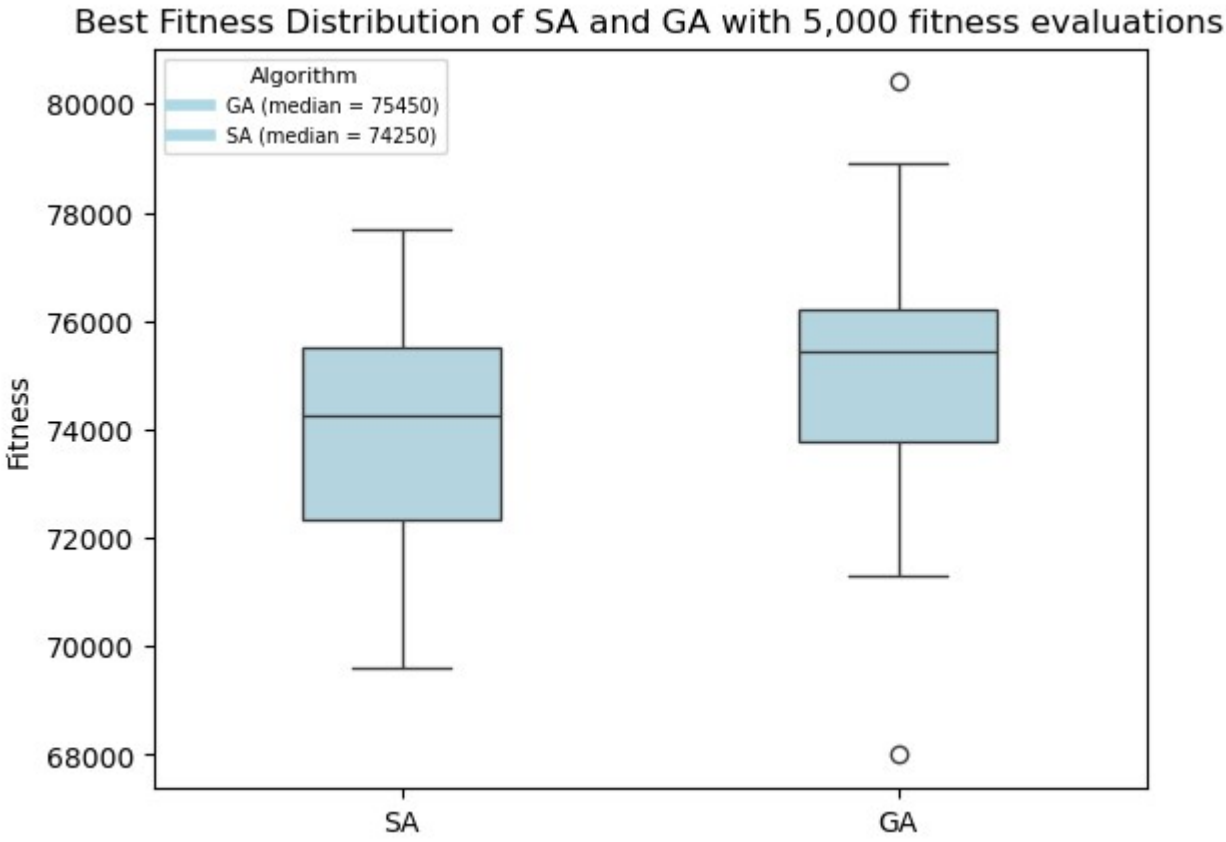


Figure 37: Best final fitness distribution after 5,000 fitness evaluations for Genetic Algorithm (GA) and Simulated Annealing (SA)