CIFO Project Report

Wedding Seating Optimization with Genetic Algorithms

Louis Berthele (20240552), Guilherme Cordeiro (20240527), Maria Radix (20240687)
Group 42

May 2025

GitHub Repository: `https://github.com/gcordeirofm/CIFO25_42`

Universidade Nova de Lisboa - Information Management School

Contents

List of Figures

List of Tables

1 Introduction

This project aimed to solve an instance of the Wedding Seat Optimization problem with the use of Genetic Algorithms.

This instance of the problem consists of distributing 64 guests of a wedding across 8 tables, with exactly 8 guests per table, where the arrangement within each table is irrelevant (i.e., doesn't matter who is seated next to who, just whether they're in the same table). The goal is to **maximize total guest happiness**, given a specific pairwise relationship matrix of all 64 guests [1], obtained through a relationship chart of all the guests [2] by attributing a numeric value to the relationship each guest has with the others [3]. (See Figure C.1 for a visualization of the guest relationships).

In order to apply genetic algorithms to this problem, and in accordance with the project requirements, we defined: the representation of an individual, and the search space associated with it; a fitness function; 3 selection methods; 3 crossover operators; 4 mutation operators. In the initial part of the report, we will define all of these and describe in detail how they work. We then detail our experimental design, analyze our results, and reflect on areas for improvement or future works.

2 Representation and Fitness Function

In order to apply genetic algorithms to this problem, we must define how to represent a solution, i.e., how to encode a seating arrangement as an individual in the algorithm's population, as well as a fitness function to quantify how good each individual is.

We must consider that solutions have to represent a complete seating arrangement that specifies which guests are seated at which table, and that they must respect a set of constraints: each table must have exactly 8 guests, and each guest must be assigned to one and only one table. Infeasible solutions that don't meet these criteria will not be explored, and our operators ensure this.

We initially considered different ways to represent a seating arrangement for the 64 guests across 8 tables:

- **Lists of guests per table**: Each table is represented as a list containing the guests assigned to it, for example: `[[0,2,4,...], [1,3,5,...], ...]`, with guest numbers ranging 0 to 63.
- **List of table assignments**: The index for each guest is assigned a table, resulting in a list of length 64, such as `[0,4,6,1,...]`, where the value at each position corresponds to the guest's table number, with table numbers ranging 0 to 7.
- **Binary encodings**: Variants where the assignment is represented using binary vectors, either per guest or per table. These were discarded due to added complexity without clear benefits.

Although the first representation is straightforward and intuitive, it suffers from high solution redundancy, as the same real-world seating arrangement can be encoded by many different orders of the table lists as well as many different orders of guests within each table list, in fact, $8!^8 \times 8! \approx 2.82 \times 10^{41}$ different representations exist for every single solution. This excessive redundancy significantly bloats the search space, possibly leading to the exploration of many equivalent solutions.

The second representation also suffers from this problem, but considerably less. This arises from the table numbering, as the same set of guests being assigned to table 1 or table 4 represents the same practical solution, leading to there being $8! = 40320$ different representations per solution. This comparison along with total search space size is presented in Table 1.

Both representations suffer from an incompatibility with most genetic operators, mainly due to the problem constraints (it is hard to combine or alter genes while guaranteeing 8 guests per table and each guest in only one table) but also this redundancy issue (reordering tables or changing their numbers makes no difference). However, we believed the second representation to be more generally compatible with a variety of crossover and mutation operators.

Considering these factors, we proceeded with the *list of table assignments* representation for this project. That is, a solution is represented by a list of 64 elements ranging 0-7, with each table number appearing exactly eight times. Considering T as the set of table numbers, $T = \{0, 1, 2, 3, 4, 5, 6, 7\}$, the search space can be defined as $S = \left\{x = (x_1 \ldots, x_{64}) \in T^{64} : \forall t \in T,\ \#\{\, i : x_i = t\} = 8\right\}$, with an incredibly large size of $|S| = \frac{64!}{(8!)^8} \approx 1.82 \times 10^{52}$.

We calculate the fitness of each seating arrangement by summing the relationship values of all guest pairs seated at the same table, and then calculating the total sum of this value across all tables. We decided on this as the best fitness function as it's aligned with the project goal by effectively reflecting total guest satisfaction in a simple and straightforward way.

3 Selection Methods and Genetic Operators

The following selection methods and genetic operators were developed or adapted in a standard genetic algorithm process, as outlined in the theoretical sessions of this course. A general overview of this process can be found in Appendix B.

3.1 Selection Methods

3.1.1 Tournament Selection. Tournament Selection chooses k random individuals from the original population to participate in a tournament. Fitnesses of each individual in the tournament are evaluated, and the individual with best fitness (highest for maximization, lowest for minimization) is selected as the winner. This individual is copied to parent population P'.

3.1.2 Ranking Selection. Ranking Selection sorts all individuals in the population based on fitness. In maximization for a population of N individuals, the one with the lowest fitness is given rank 1, and the one with the highest is given rank N (the reverse is true in minimization). Selection probabilities are then calculated based on rankings; better ranked individuals have higher probability of being selected and vice-versa. Given a population P of N individuals with individual fitness values f_i and fitness ranks r_i, selection probabilities are calculated according to the following formula:

$$P(\textit{select individual } i) = \frac{r_i}{\sum_{j=1}^{N} r_j}, \forall\, i \in \{1, 2, 3, ..., N\} \tag{1}$$

This method disregards the magnitude of difference in fitnesses of individuals, relying only on their relative values.

3.1.3 Fitness Proportionate Selection. The same selection method implemented in practical classes. The method sums the fitness values of the population, and assigns a selection probability equal to an individual's share of the total fitness value. Given a population P of N individuals with individual fitness values f_i, selection probabilities are calculated according to the following formula.

$$P(\textit{select individual } i) = \frac{f_i}{\sum_{j=1}^{N} f_j}, \forall\, i \in \{1, 2, 3, ..., N\} \tag{2}$$

3.2 Crossover Operators

For all Crossover Operators, our biggest conceptual hurdle in creating a valid offspring was how to maintain exactly 8 instances of the 8 table indices. Methods that preserve a single instance of each index, such as Cycle and Partially Matched Crossover, seemed to be a step in this direction. However, they were not functional in their baseline forms due to the fact that, in our solution representation, each table index has multiple instances. To solve this, we developed a way to temporarily transform our table indices into unique values by "tagging" each instance of a table index with its occurrence index. For example, the first occurrence of table index 2 in a solution becomes $(2, 0)$, the second occurrence becomes $(2, 1)$, and so on and so forth. (Conceptually, this can be thought of as representing a seat number at a given table). This "tagging" transforms all duplicate table indices into unique tuples, which are fully functional with the Cycle, Partially Matched, and other crossover methods designed to preserve index instances.

In all crossover methods implemented in our genetic algorithm, solution representations are tagged, crossover is applied, and solutions are untagged and returned in their original representation form.

3.2.1 Cycle Crossover. A cycle begins by selecting a random index and its corresponding value (a unique $(table, occurrence)$ pair) from tagged Parent 1, which is copied into Offspring 1. We then find the index of this value in Parent 2, and the cycle returns to Parent 1 with the new index. This process continues until we arrive back at the original random index, at which point all remaining values are copied from Parent 2. The reverse is done for Offspring 2. When both offspring have been filled, the tags are removed and they are returned in their original representation format.

3.2.2 Partially Matched Crossover. In Partially Matched Crossover, two random indices are selected to create a "window" in each parent solution. Values within the window are copied to the opposite offspring (i.e. Parent 1 copied into Offspring 2). Remaining indices are filled with values from the other parent, in the order that they appear, skipping values already present in offspring from window swap.

3.2.3 Position Based Crossover. Position Based Crossover operates in a similar manner to Partially Matched crossover, except that a random amount of indices are selected and exchanged, rather than a window. The method selects a random set of indices from both parent solutions, then copies the values in each parent to one of the offspring (i.e. Parent 1 values copied to Offspring 1), maintaining the indices. The remaining indices are filled with values from the other parent, in the order in which they appear, skipping values already present from the previous swap. See Appendix C.2 for an illustration with explanation. [4]

3.3 Mutation Operators

3.3.1 Block Swap Mutation. Block swap randomly selects two segments of a given size from the solution representation and swaps all values in the two blocks. [4]

3.3.2 Shuffle Subsequence Mutation. Shuffle Subsequence is a modification of Inverse Subsequence Mutation presented in practical classes. Shuffle Subsequence mutation selects two random indices from the solution, and randomly shuffles the values between them. [4]

3.3.3 N-Swap Mutation. N-Swap Mutation is a modification of baseline Swap Mutation presented in practical classes. N-Swap mutation randomly selects n pairs of indices from given solution, and swaps their values, where n is a parameter of the mutation function.

3.3.4 Displacement Mutation. Displacement Mutation selects two random indices from a given solution representation. The subsequence of values between these two indices is removed from the solution, and reinserted at another random index. [4]

4 Experimental Setup and Performance Analysis

4.1 Experimental Setup

After developing a set of crossover, mutation, and selection operators, we set up an experimental framework that allowed us to explore different combinations of operators, to tune hyperparameters, and to find the configuration that works best for our problem. The 10 operators we developed are outlined in an earlier chapter of this report. The hyperparameters include: crossover probability, mutation probability, population size, generations, number of runs, elitism (boolean), number of swaps (specific to N-Swap Mutation), and block size (specific to Block Swap mutation).

Combining and running all combinations of operators and hyperparameter values simultaneously would have required excessive computational time, which was not feasible within the scope of this project. In order to run a substantial amount of combinations, we decided to divide the experiment into two phases.

The goal of Phase 1 was to determine which combination of operators and selection methods resulted in the highest fitness. Mutation probability, crossover probability, and all other global hyperparameters were set to values taken from examples given during the practical classes, and mutation-specific parameters were set at aggressive values in an attempt to avoid local optima and fully explore the search space. [4]. All hyperparameters were kept constant to test all operator configurations and therefore isolate their effects. (See Appendix A.2 for default parameters of Phase 1). This resulted in a total of 3 (crossover) $\times$ 4 (mutation) $\times$ 3 (selection) = 36 configurations to be run. The best performing combination was then taken as a base configuration for Phase 2. Additionally, to assess if any of the Average Best Fitness (ABF) values obtained from the configurations were statistically significantly different than the others, we performed an ANOVA difference in means test and post-hoc pairwise t-tests. Phase 1 was run two times, and the results were concatenated in order to increase the number of observations and thus suitability for statistical testing.

In Phase 2, we ran a gridsearch iterating through a range of hyperparameter value combinations on the best performing configuration found in Phase 1. Due to time and computational resource restrictions, we decided to limit the grid search to the following hyperparameters:

- Crossover Probability: tested one value randomly selected from each of the intervals: $[0, \frac{1}{5}], [\frac{1}{5}, \frac{2}{5}], ...[\frac{4}{5}, 1]$
- Mutation Probability: tested one value randomly selected from each of the intervals: $[0, \frac{1}{5}], [\frac{1}{5}, \frac{2}{5}], ...[\frac{4}{5}, 1]$
- n from N-Swap Mutation: tested $n = 3$ and $n = 6$ (lower values than Phase 1 for a less disruptive mutation operation)

This yielded a total of $5 \times 5 \times 2 = 50$ hyperparameter combinations on our chosen configuration. Population size, generations, and number of runs were kept constant due to computational and time limitations. (See Appendix A.3 for hyperparameter values). Elitism was fixed as True, both for the same practical reasons and to add resilience in our population to the more aggressive mutation operations we tested. Block size was not included as the Block Swap mutation operator was not part of the best performing configuration from Phase 1. As with the results from Phase 1, an ANOVA test was performed on the mean fitness values of each configuration, and the top performing configuration was tested for statistical significance in a pairwise t-test against all other configurations.

4.2 Performance Analysis

Performance analysis of the configurations of our genetic algorithm was carried out in two ways: informally, in an observational-style approach, and formally, in statistical tests. Our statistical analysis involves an ANOVA difference in means test, as well as a series of pairwise t-tests on our best configurations against all other configurations at both stages. Conceptual details and statistical assumptions necessary to perform these tests can be found in Appendix D.

4.2.1 Phase 1. As mentioned, analysis of configuration performance in Phase 1 was based on aggregated results of two trials, 30 runs each. Our best performing configuration included the following combination of operators: **Cycle Crossover, N-Swap Mutation** and **Ranking Selection**, with an average final generation fitness of **47586.67**. In the sections that follow, this configuration is referred to as ***Phase 1 Best***.

From examining ABF boxplot distributions of all tested configurations, we noticed that those utilizing Block Swap Mutation performed worse than other mutation methods. In contrast, Tournament Selection and Ranking Selection outperformed Fitness Proportionate Selection. Visualizations of distributions of fitness results from Phase 1 can be found in Appendix C.3.

An ANOVA test of difference in mean fitnesses of our configurations resulted in a p-value of 0.00, indicating with high statistical significance that the ABF in one of the configurations is different than the others. To determine specifically if *Phase 1 Best* was significantly different, we ran a one-sided pairwise t-test between *Phase 1 Best* and all other configurations. Results from this test indicate that the ABF of the final generation in *Phase 1 Best* is statistically significantly higher than all but six other tested configurations. Interestingly, the configurations of the top competitors are diverse: all crossover operators are represented, as well as all mutation operators except Block Swap. Given more time, we would have explored hyperparameter tuning on these six additional configurations. Despite the lack of total statistical significance, we continued our experiment with *Phase 1 Best*.

4.2.2 Phase 2. The best resulting configuration of hyperparameters tested in Phase 2 was: **crossover probability: 0.92, mutation probability: 0.25, n in N-Swap Mutation: 3**, with an average final generation fitness of **60546.67**. Moving forward, this configuration will be referred to as *Phase 2 Best*. Visualizations of fitness evolution and fitness values across runs for *Phase 2 Best* can be found in Appendix C.5.

Analysis of boxplot distributions of Phase 2 results showed configurations where $n = 6$ (6 pairs of swaps in N-Swap mutation) performed significantly worse than configurations where n = 3. Additionally, configurations with high mutation probabilities (0.77 and 0.94) performed worse than those with lower mutation probabilities. Both of these observations are likely due to the fact that a higher n and a higher mutation probability increase the chances that valuable traits will be destroyed by mutation and not carried on into a new population. Configurations with mutation probability of 0.25 have the overall highest performance; this level seems to be valuable in introducing diversity to our solution population without being overly disruptive. Finally, we noticed crossover probabilities do not appear to have a significant impact on the distribution of fitness values.

From a statistical standpoint, results of Phase 2 were similar to Phase 1; ANOVA indicated the presence of at least one statistically significantly different ABF, and pairwise t-tests revealed *Phase 2 Best* was statistically higher than all but four other configurations. Three out of four of the other top performers have exactly the same configuration as *Phase 2 Best*, but with variable crossover probabilities. This aligns with our observations from the boxplots, which indicated crossover probability did not have a noticeable effect on fitness distributions. In fact, the single best individual we obtained (fitness **68,000**) had a configuration similar to *Phase 2 Best* but with a crossover probability of 0.47, further confirming our observations.

5 Review and Outlook

We saw an increase in average final generation fitness of roughly 12,000 units between Phase 1 and Phase 2 - an over 25% increase. This jump is partially attributed to success in our hyperparameter tuning phase, and partially due to the fact that generations increased from 100 to 150 from Phase 1 to Phase 2. Examining our fitness evolution curves revealed our configurations have yet to converge on a single highest fitness value - we have yet to reach the global optimum. Given more time and computational resources, we would have increased both the size of our initial population and the number of generations in each run, in the hopes that this would bring us to the global maximum fitness value.

References

[1] I. M., *Pairwise relationship matrix*, https://liveeduisegiunl-my.sharepoint.com/:x:/g/personal/imagessi_novaims_unl_pt/EYTlb599xcZIlhLRp_lBUpUBdsxsFGoketsp5AcX1NuUew?e=fvOLyn, Accessed: 2025-05-22, 2024.

[2] I. M., *Chart of guests and their relationships*, https://liveeduisegiunl-my.sharepoint.com/:i:/g/personal/imagessi_novaims_unl_pt/EYUUGfopZVhLlikDRC_tKyUB7xdHLwzkT8mlsE3oO7_40g?e=SLT2F1, Accessed: 2025-05-22, 2024.

[3] I. M., *Relationship value table*, https://liveeduisegiunl-my.sharepoint.com/:i:/g/personal/imagessi_novaims_unl_pt/EbMQbdPBkBtImfdBuy5MWDwBg23BGmflfRnFDm5uF0R7ww?e=DUNCXd, Accessed: 2025-05-22, 2024.

[4] P. Larranaga, C. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic, "Genetic algorithms for the travelling salesman problem: A review of representations and operators," *Artificial Intelligence Review*, vol. 13, pp. 129–170, Jan. 1999. DOI: 10.1023/A:1006529012972.

A Configuration Details

A.1 Representation Comparison

Table 1: Redundancy comparison between the two candidate representations

Note: The "Representation" column considers an example with 6 guests across 2 tables

Format	**Representation**	**# Rep. per solution**	**Search space size**	**Pros**	**Cons**
Guests per table	`[[1,3,5], [2,4,6]]`	$8!^9 \approx 2.82 \times 10^{41}$	$64! \approx 1.27 \times 10^{89}$	Intuitive	Really redundant
Guests' table num.	`[1,2,1,2,1,2]`	$8! = 40320$	$\frac{64!}{(8!)^8} \approx 1.82 \times 10^{52}$	Less intuitive	Less redundant

A.2 Phase 1 Default Hyperparameters

- Crossover Probability: 0.8
- Mutation Probability: 0.2
- Number of Runs: 30 (ran twice, so 60 runs in total)
- Population Size: 50
- Generations: 100
- `Elitism = True`
- Operator-specific hyperparameters:
  - n in N-Swap Mutation: 8
  - `block_size` in Block Swap Mutation: 4

A.3 Phase 2 Default Hyperparameters

- Number of Runs: 30
- Population Size: 50
- Generations: 150
- `Elitism = True`

B Genetic Algorithm Pseudo Code

Algorithm 1 Genetic Algorithm

Initialize a population P of N individuals (at random)
while termination condition not met (e.g., max generations) **do**
  Create an empty population P'
  if elitism is used **then**
    Insert the best individual from P into P'
  end if
  while $|P'| < N$ **do**
    Select 2 individuals from population P using a selection algorithm
    Choose an operator between crossover and replication with probabilities P_c and $1 - P_c$, respectively
    Apply the chosen operator to generate offspring
    Apply mutation to offspring with mutation probability P_m
    Insert mutated offspring into P'
  end while
  Replace P with P'
end while
return the best individual in P

C Visualizations

C.1 Visualization of Guest Relationships

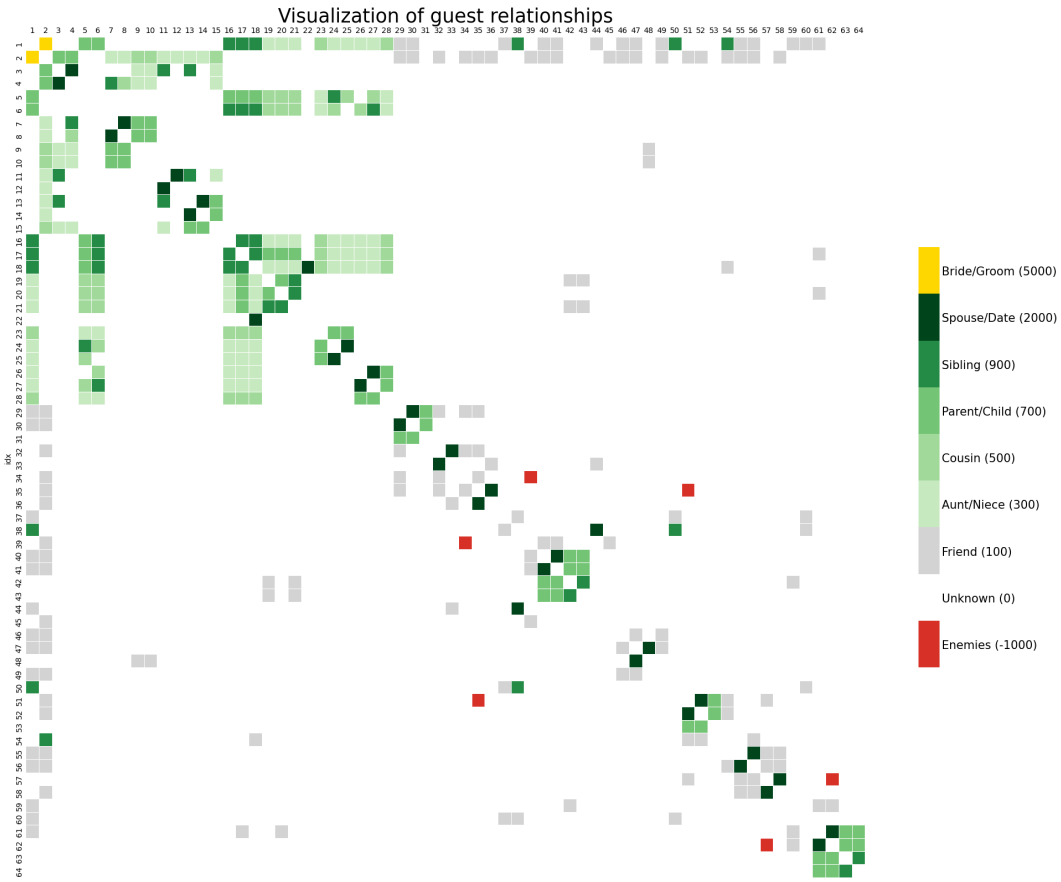

Figure 1: Heatmap representing Pairwise Relationship Values between Guests

C.2 Position Based Crossover

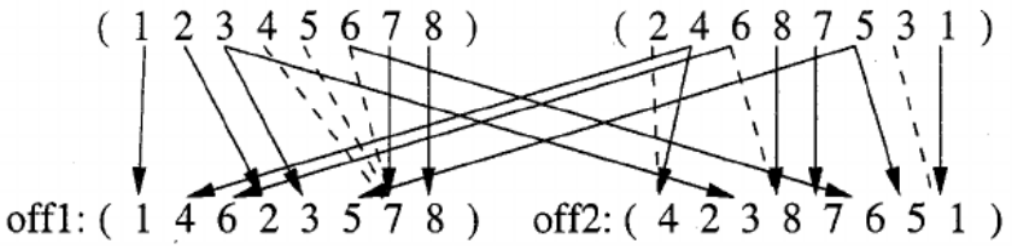

Figure 2: Position Based Crossover from Larranaga et al. [4]

"The position-based operator also starts by selecting a random set of positions in the parent tours. However, this operator imposes the position of the selected cities on the corresponding cities of the other parent. For example, consider the parent tours (12345678) and (24687531), and suppose that the second, third, and the sixth positions are selected. This leads (Figure 2) to the following offspring: (14623578) and (42387651)." — from Larranaga et al. [4] chapter 4.3.5.

C.3 Boxplots of Phase 1 Fitness Distributions among Configurations

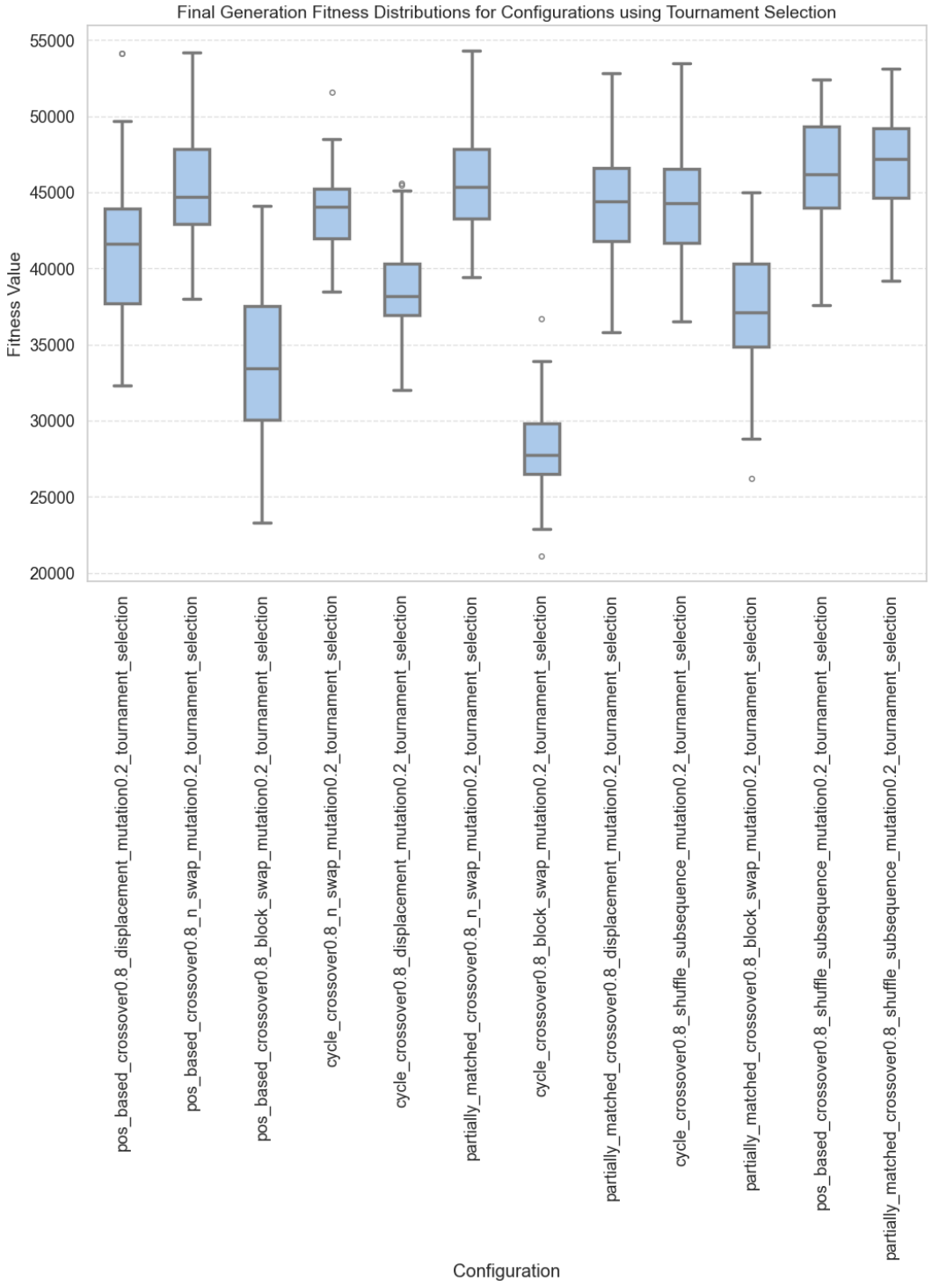

Figure 3: Boxplots of Phase 1 Fitness Distributions for Tournament Selection

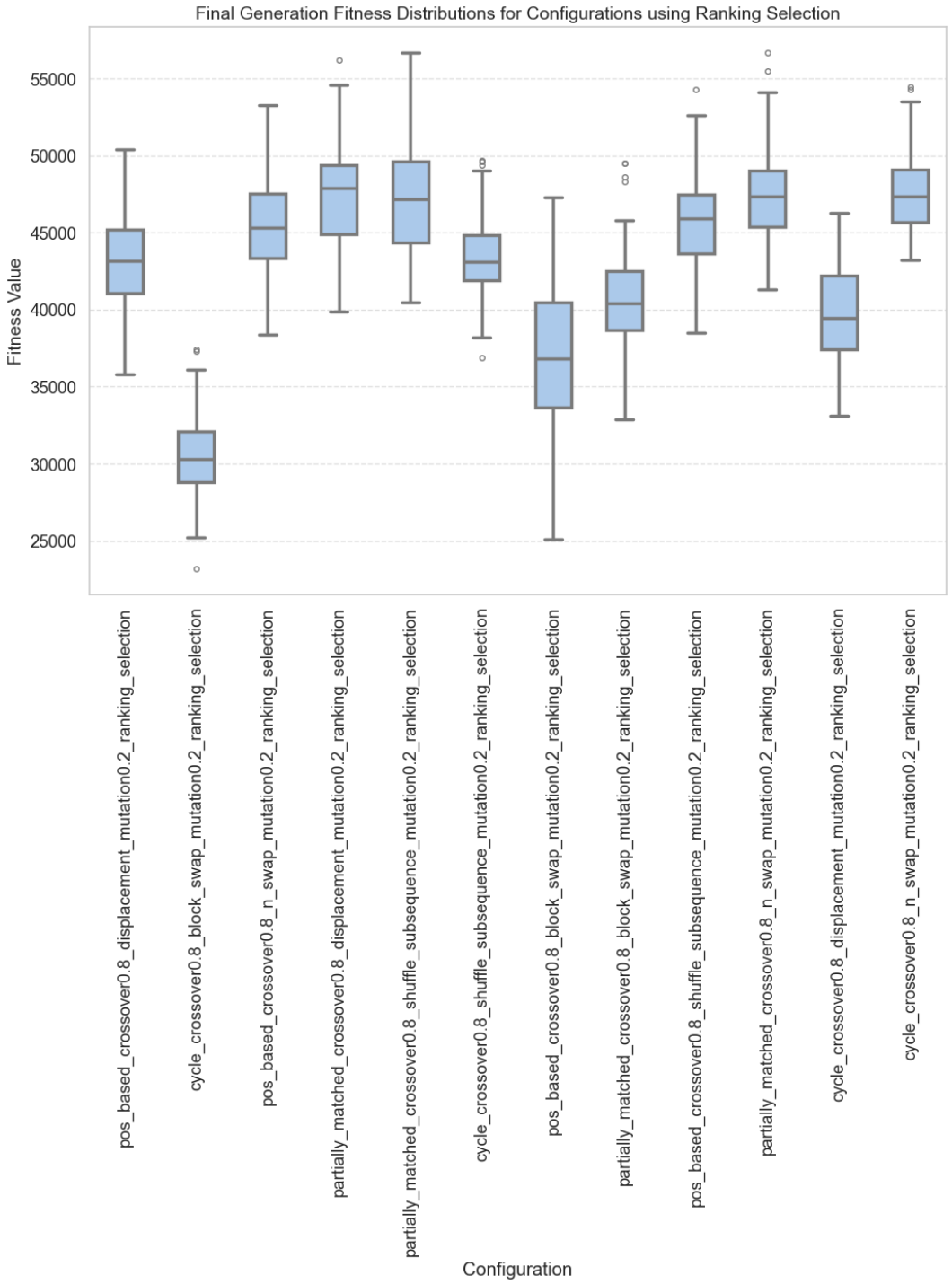

Figure 4: Boxplots of Phase 1 Fitness Distributions for Ranking Selection

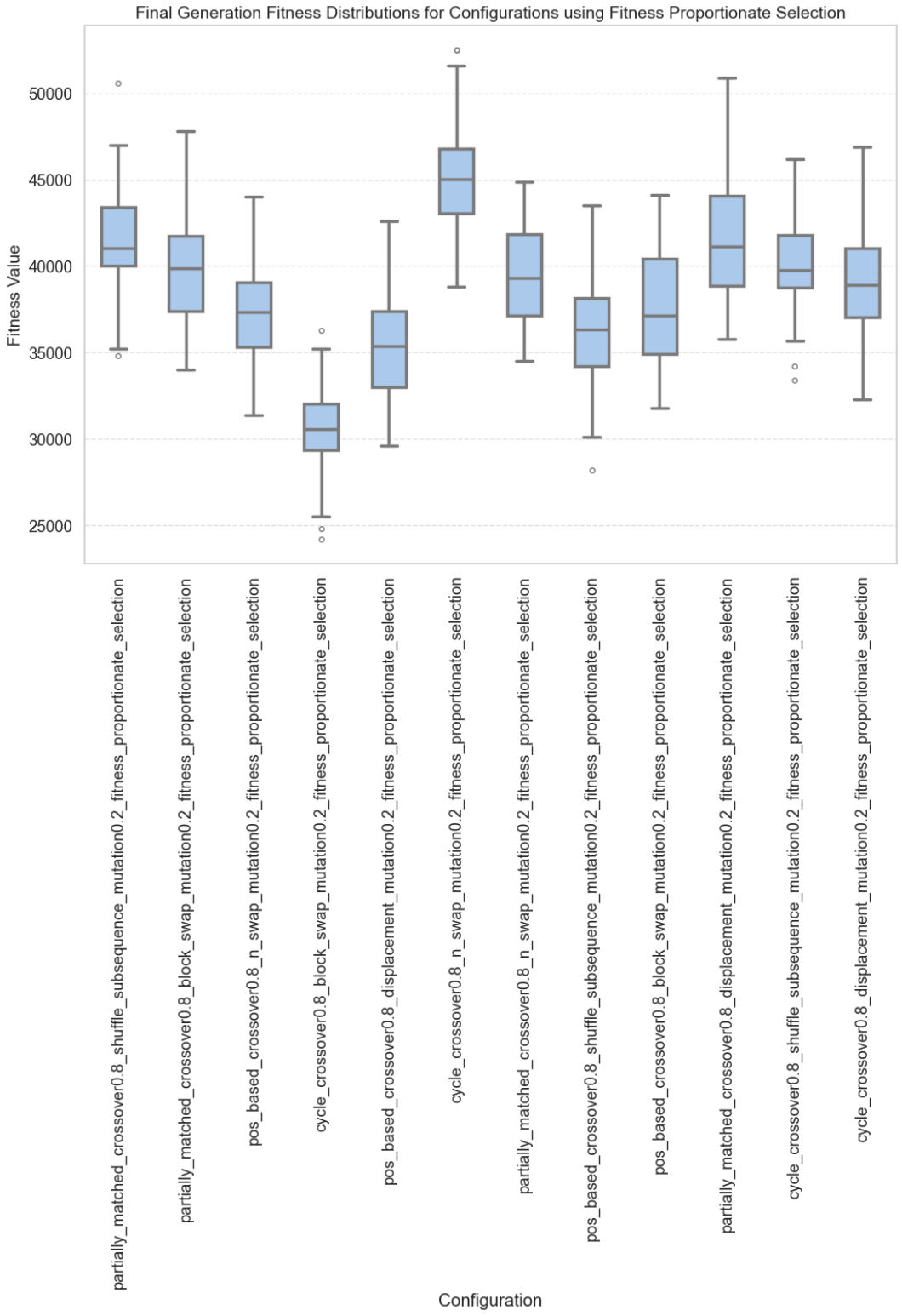

Figure 5: Boxplots of Phase 1 Fitness Distributions for Proportionate Selection

C.4 Boxplots of Phase 2 Fitness Distributions among Configurations

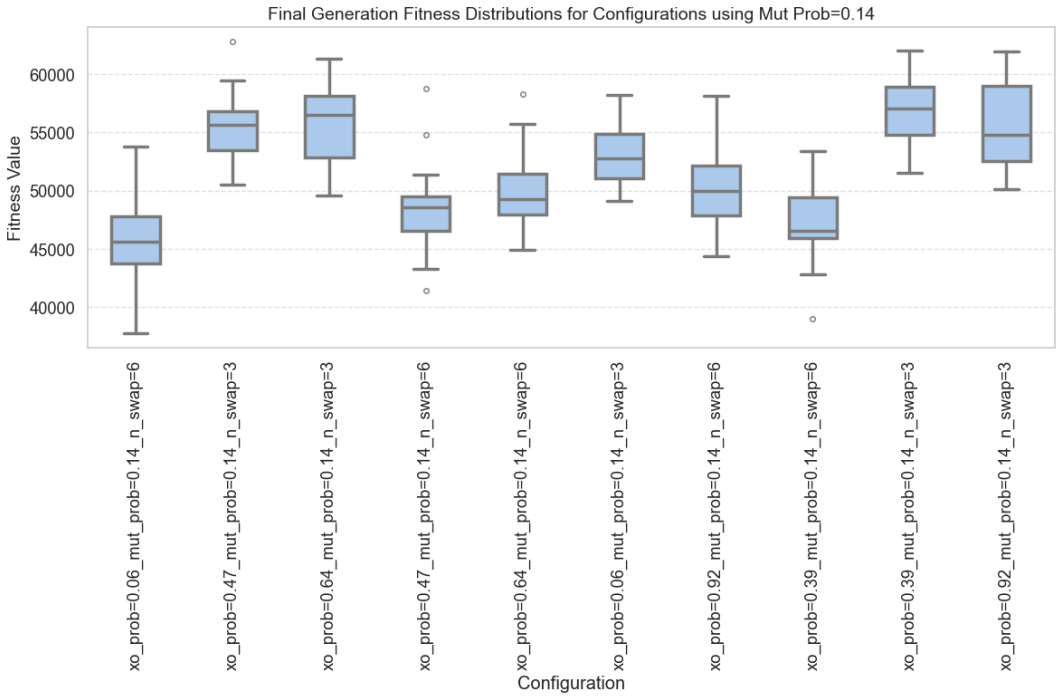

Figure 6: Boxplots of Phase 2 Fitness Distributions for Mutation Probability 0.14

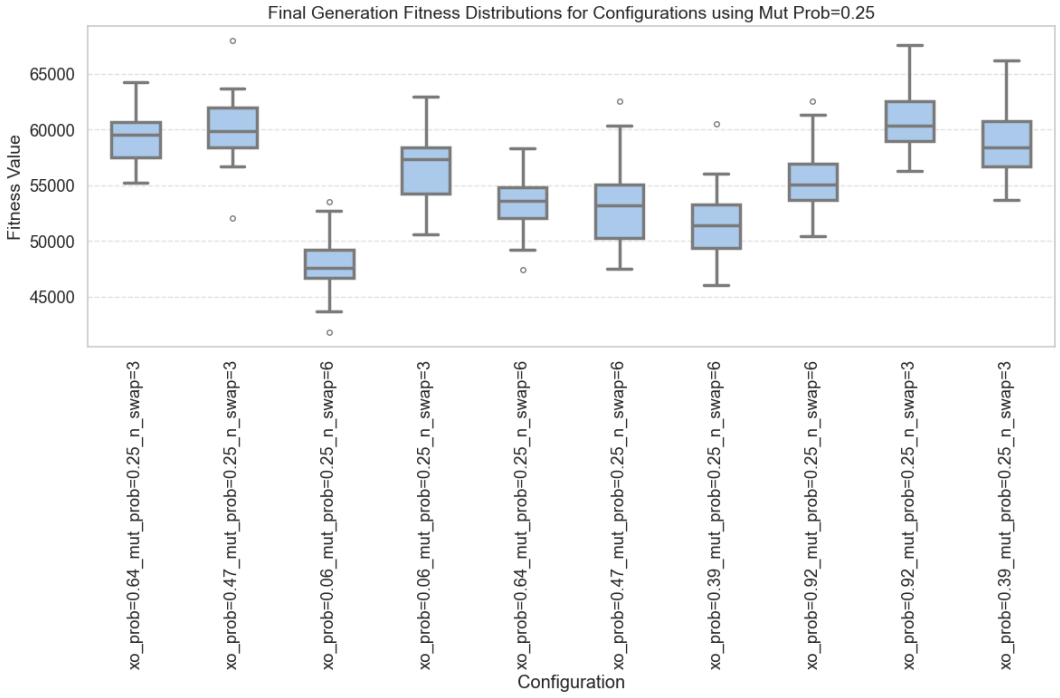

Figure 7: Boxplots of Phase 2 Fitness Distributions for Mutation Probability 0.25

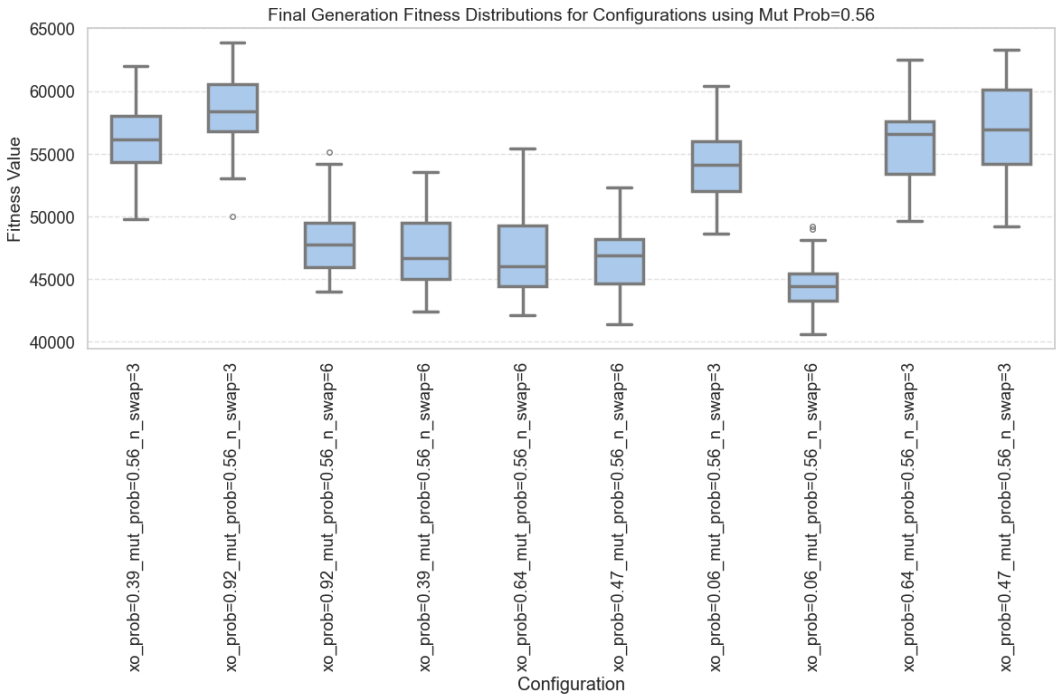

Figure 8: Boxplots of Phase 2 Fitness Distributions for Mutation Probability 0.56

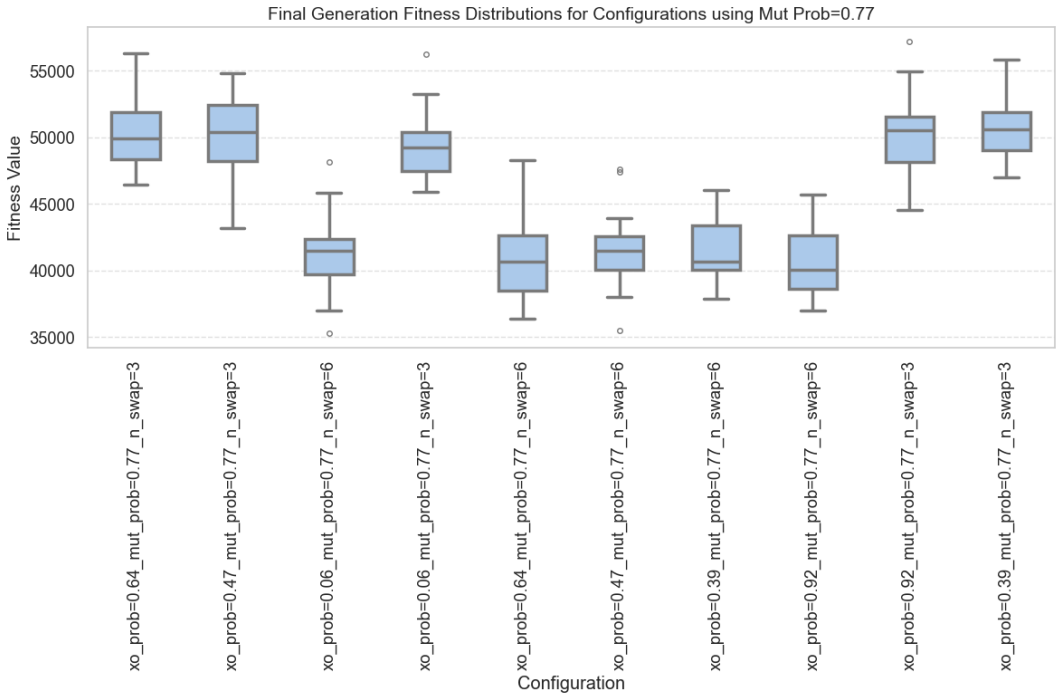

Figure 9: Boxplots of Phase 2 Fitness Distributions for Mutation Probability 0.77

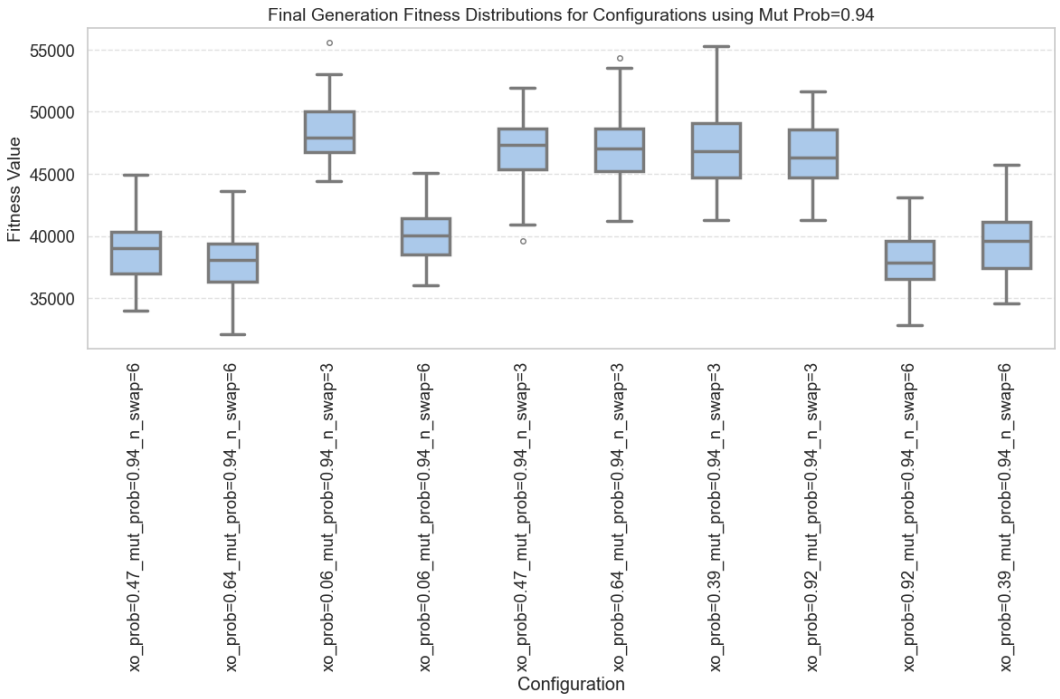

Figure 10: Boxplots of Phase 2 Fitness Distributions for Mutation Probability 0.94

C.5 Phase 2 Best Configuration Fitness Evolution and Distribution

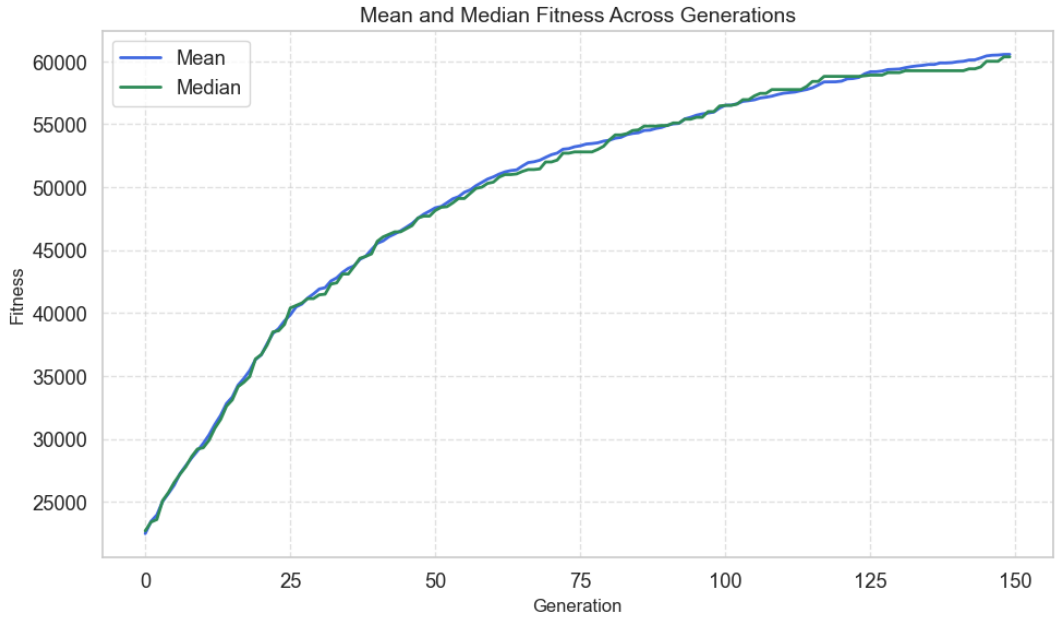

Figure 11: Phase 2 Best Configuration Fitness Evolution

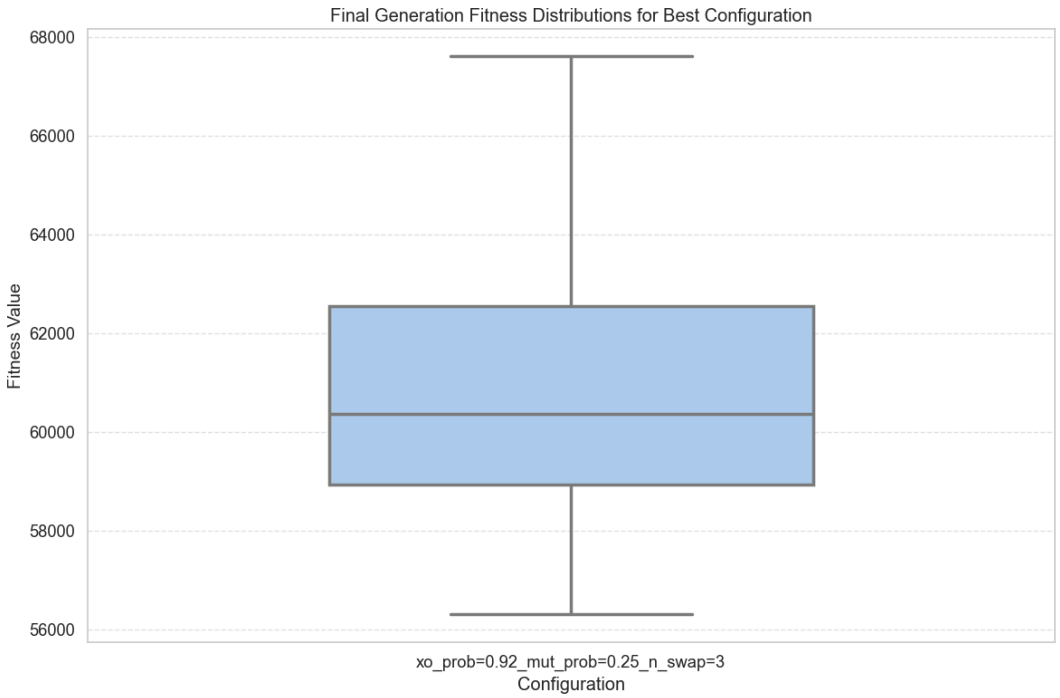

Figure 12: Phase 2 Best Configuration Fitness Distribution

C.6 Phase 2 Top 8 Configuration Fitness Evolution

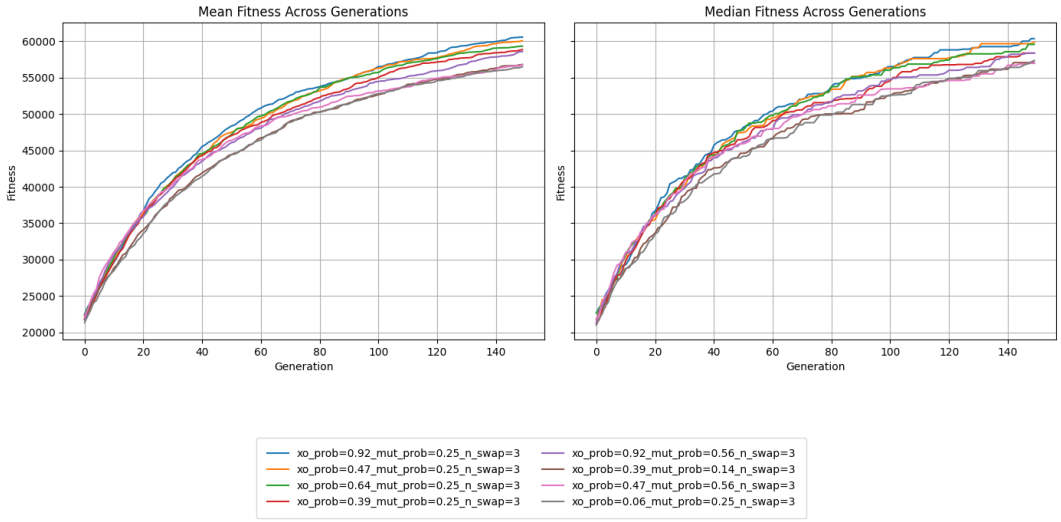

Figure 13: Phase 2 Top 8 Configuration Fitness Evolution

D Statistical Tests

D.1 ANOVA

ANOVA is used to test if variation of means **between** groups is large compared to variation of means **within** groups. Hypotheses are as follows:

- $H_0 : mean_1 = mean_2 = mean_3 = ... = mean_n, \forall i \in \{1, \ldots, n\}$ where each $mean_i$ is the mean of a configuration)
- H_a: at least one mean is different from the others

To test these hypotheses, an ANOVA test compares the Mean Squared Error (MSE; weighted average of within-group variability) with the Mean Squared Treatments (MST; weighted average of between-group variability). The test calculates an F-statistic of $F_s = \frac{MST}{MSE}$. If $MST >> MSE$, then F_s is big, we reject H_0, and there is a large difference in the means of the groups. On the contrary, if $MST << MSE$, then F_s is small, we fail to reject H_0, and there is not much difference between the means of the groups. Results of this test simply indicate **if** one of the means in the group is significantly different from the others. A post-hoc test is needed to determine **which** of the means it is.

D.2 Pairwise T-Test

To test the statistical significance of a particular mean, pairwise t-tests are run between the mean of the group in question and all other groups. A p-value is returned for each pair of means, indicating if the mean of the target group is statistically significantly different in each pairing. To account for the increased probability of making a Type I error (rejecting the null hypothesis when it is true) in repeated pairwise tests, a lower p-value is necessary for a given statistical significance level. For a 5 percent significance level, the p-value becomes: $\frac{0.05}{n}$, where n is the number of pairwise tests conducted. For Phase 1, this means the p-value must be $\leq \frac{0.05}{36} = 0.000138$ for significance, and $\leq \frac{0.05}{50} = 0.0001$ for Phase 2.

D.3 Assumptions

Both ANOVA and the pairwise t-tests can only be carried out if certain statistical assumptions are met. See below for assumptions and their validity in the case of our experiment.

- Groups are independent: Yes. Configurations have randomly initialized populations and distinct combinations of operators and hyperparameters.
- Observations are random samples: Yes. We are using the entire population (census) of each configuration's output. A census satisfies this condition, as well.
- Group population distributions are approximately normal: Yes. Safe to assume because for each configuration, $n \geq 30$, ($n = 60$ for Phase 1, and $n = 30$ for Phase 2).
- Group standard deviations are approximately equal: Yes. Boxplots were examined to make an informal, observational decision about standard deviations. Distributions were normal and IQRs were approximately equal, so we can reasonably assume standard deviations are, as well.