CIFO PROJECT

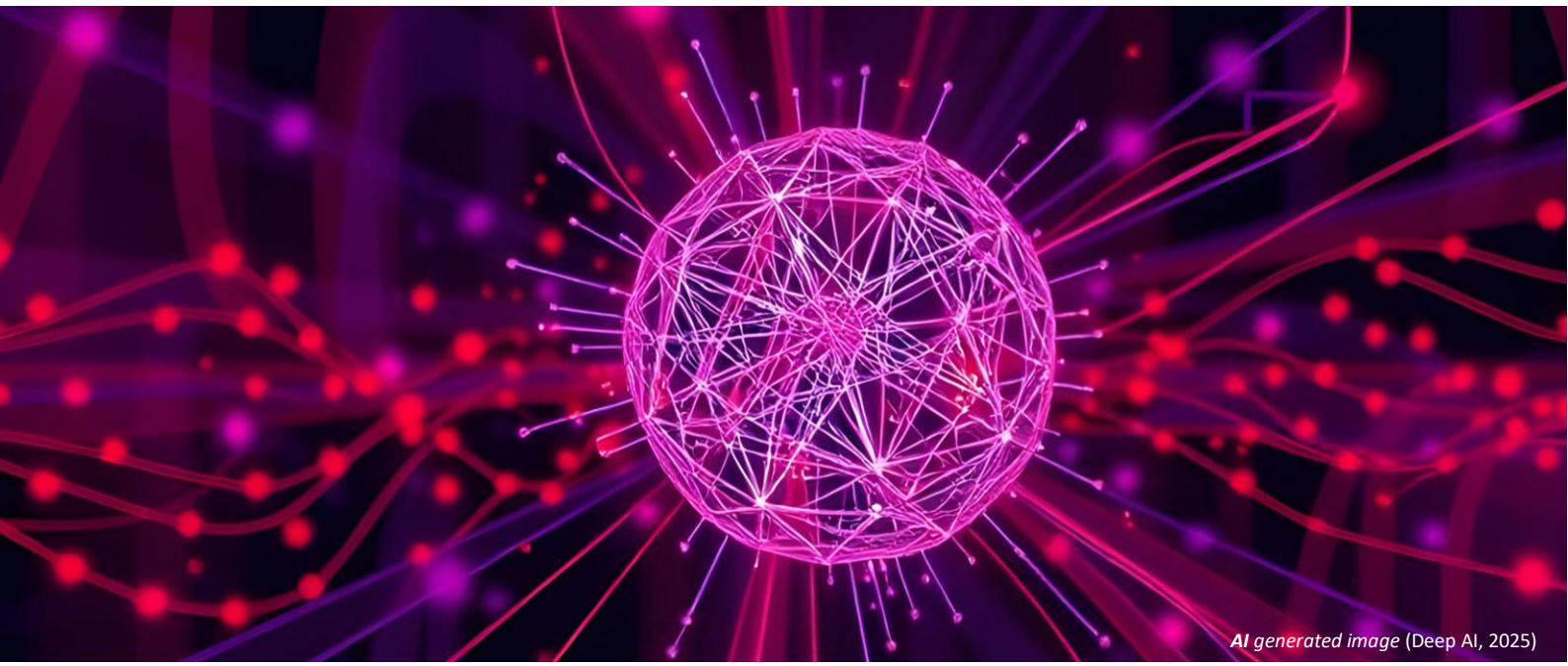
Master in Data Science and Advanced Analytics

NOVA Information Management School

Universidade Nova de Lisboa

Music Festival Lineup Optimization

Genetic Algorithms



AI generated image (Deep AI, 2025)

Group W

André Silvestre, 20240502

Filipa Pereira, 20240509

Umeima Mahomed, 20240543

Fall/Spring Semester 2024-2025

TABLE OF CONTENTS [1]

[1] **GitHub Repository Link:** https://github.com/Silvestre17/CIFO_Project_GroupW

ABSTRACT

This study optimizes music festival lineups using Genetic Algorithms (GAs) with a 2D schedule representation and a multi-objective fitness function balancing artist popularity, genre diversity, and fan conflict. Testing 24 GA configurations, Rank Selection with Cycle Crossover outperformed, matching Simulated Annealing and surpassing Hill Climbing. Future enhancements include tuning parameters and exploring Pareto-based GAs for diverse solutions.

Keywords: Multi-Objective Optimization, Music Festival Scheduling, Metaheuristics, Genetic Algorithms, Hill Climbing , Simulated Annealing

---

1. INTRODUCTION

Organizing a music festival lineup is a hard task that involves scheduling many artists across limited stages and time slots, while balancing competing goals. As the number of artists increases, scheduling complexity grows exponentially—in our case, 35 artists in 35 slots yield $35! \approx 1.03 \times 10^{40}$ combinations. Metaheuristic algorithms are well-suited to this, efficiently exploring vast search spaces. This project compares three such algorithms: **Genetic Algorithms** (GAs) (evolve populations of solutions), **Simulated Annealing** (SA) (probabilistic exploration to escape local optima), and **Hill Climbing** (HC) (greedy local search). The ***No Free Lunch Theorem*** motivates this comparative approach, as no single optimization algorithm is best for all problems. The stochastic nature of these methods necessitates experimental comparison (30 samples of each) to determine their effectiveness for this specific **scheduling problem**. [1][2][4]

The literature (**Table A1**) [3][4][5] explain that GAs use operators like PMX, OX, and CX for crossover, and Swap or Insertion for mutation, to handle specific parts of the problem, such as arranging artists or stages. Selection methods like tournament or roulette help find good solutions. Research [6] compares GAs, SA, and HC. Both GAs and SA slowly get closer to the best solution over time, as proven by the ***Theorem of Asymptotic Convergence for GAs*** and similar properties for SA, making them great for tricky scheduling tasks.

The report is structured as follows**: Section 2** explains the GA methodology, including problem definition, individual representation, fitness function, genetic operators, and hyperparameter tuning. **Section 3** analyses experimental results, operator effects, elitism, and compares GA with HC and SA using statistical tests. **Section 4** concludes with key insights and future research directions in event scheduling optimization.

2. GENETIC ALGORITHM

2.1. Definition of the Problem

The aim is to create an optimal music festival schedule by assigning **35 artists** to **5 stages**, each with **7 time slots**, making 35 unique performance spots. The goal is to balance three equally weighted objectives:

1. **Maximization of Prime Slot Popularity:** Scheduling artists with high popularity scores in the final time slot of each stage.
2. **Maximization of Genre Diversity:** Promoting a varied distribution of musical genres across the stages within each concurrent time slot.
3. **Minimization of Fan Conflict Penalty:** Reducing scheduling conflicts that arise when artists sharing significant fan bases perform at the same time on different stages.

The algorithm uses two data sources: *artists(in).csv*, which lists artist names, genres, and popularity scores ($0 - 100$), and *conflicts(in).csv*, a matrix showing fan base conflict scores ($0 - 1$) between artists.

Our GAs (**Figure B1**) are designed to work for any number of stages, slots, and artists, as long as the required data files are provided and meet the problem's constraints.

2.2. Individual Representation

A key design choice that makes our code adaptable for different festival sizes is how we represent a lineup (genotype). Each potential festival schedule, or individual solution, is encoded as a *2D NumPy array*. The array's dimensions match the number of stages and time slots. Each cell in this grid $(stage, slot)$ holds a unique ID for an artist, as shown in **Figure C1**. We chose this matrix format because it's intuitive and makes it easy to check artists by time slot (columns) or by stage (rows), which is essential for fitness (phenotype).

This representation is part of a *Python* class structure where specialized solutions for GAs (*LineupSolutionGA*), HC (*LineupSolutionHC*), and SA (*LineupSolutionSA*) all build upon a common *LineupSolution* base. This structure is outlined in the code schema (**Figure D1**).

While the 2D matrix is how we think about the schedule, it is more practical to work on a 1D list (a flattened version of the matrix) for the crossover operator. So, for this, we temporarily flatten the 2D schedule into a single list of artist IDs. The operator acts on this list, and then we reshape it back into the 2D grid. This lets us use powerful, well-known permutation operators while keeping the schedule easy to understand in its 2D form.

2.3. Fitness Function

The fitness function (implemented in *LineupSolution*) evaluates the quality of a candidate lineup and guiding the metaheuristic algorithms' search process. Since the music festival scheduling problem involves multiple objectives, the fitness function combines three key criteria—prime slot popularity (**pop**), genre diversity (**div**), and fan conflict (**con**)— into a single scalar value $(0-1)$. These objectives are considered equally important and are therefore assigned equal weights in the overall fitness calculation.

The overall fitness $\boldsymbol{F}$ for a given lineup $\boldsymbol{L}$ is calculated as follows:

$$F(L) = \frac{1}{3}\cdot\underbrace{\frac{\sum_{s=0}^{S-1}\ \text{popularity}\left(L_{s,T-1}\right)}{\text{MaxPrimePop}}}_{N_{\text{pop}}(L)} + \frac{1}{3}\cdot\underbrace{\frac{1}{T}\sum_{t=0}^{T-1}\frac{\left|\{\text{genre}\left(L_{s,t}\right)\mid s=0,\dots,S-1\}\right|}{\text{MaxPossibleUniqueGenresPerSlot}}}_{N_{\text{div}}(L)} + \frac{1}{3}\cdot\underbrace{\left(1-\frac{1}{T}\sum_{t=0}^{T-1}\frac{\sum_{s_1=0}^{S-2}\sum_{s_2=s_1+1}^{S-1}C\left(L_{s_1,t},L_{s_2,t}\right)}{\text{WorstCaseSlotConflict}}\right)}_{1-N_{\text{con}}(L)}$$

Specifying for our problem:

$$F(L) = \frac{1}{3}\cdot\underbrace{\frac{\sum_{s=0}^{4}\ \text{popularity}\left(L_{s,6}\right)}{500}}_{N_{\text{pop}}(L)} + \frac{1}{3}\cdot\underbrace{\frac{1}{7}\sum_{t=0}^{6}\frac{\left|\{\text{genre}\left(L_{s,t}\right)\mid s=0,\dots,4\}\right|}{5}}_{N_{\text{div}}(L)} + \frac{1}{3}\cdot\underbrace{\left(1-\frac{1}{7}\sum_{t=0}^{6}\frac{\sum_{s_1=0}^{3}\sum_{s_2=s_1+1}^{4}C\left(L_{s_1,t},L_{s_2,t}\right)}{10}\right)}_{1-N_{\text{con}}(L)}$$

Where $\boldsymbol{s}$ represents the stage index and $\boldsymbol{t}$ represents the time slot index.

Each component represents the following:

1. **Normalized Prime Slot Popularity (N_{pop}):** The sum of popularity of artists performing in the prime time slots (i.e., the last slot on each stage), normalized by the maximum possible value — assuming each prime slot is occupied by an artist with the highest popularity score (e.g., for 5 stages, $max = 5 \times 100 = 500$).
2. **Normalized Genre Diversity (N_{div}):** The sum of the number of unique genres present in each time slot across stages, normalized by the number of stages. The final value is the average of these normalized counts across all time slots.
3. **Normalized Conflict Penalty (N_{con}):** The sum of the conflict scores between all artist pairs performing at the same time on different stages, normalized by the worst-case total — assuming every possible pair has the highest conflict score (e.g., for 5 stages, $max = 1 \times C_2^5 = 10$). The final value is the average of these normalized counts across all time slots.

Note that $N_{con}(L)$ measures fan conflict as a penalty, where lower values are better. To align this with the goal of maximizing fitness, the term is inverted as $1 - N_{con}(L)$.

Each component score – $N_{pop}(L)$, $N_{div}(L)$ and $N_{con}(L)$ – is normalized to the range $[0,1]$ using Min-Max scaling. This ensures fair contribution from each objective regardless of their original scales, based on theoretical best- and worst-case values. To better understand how our fitness works we leave an example in **Figure E1**.

2.4. Genetic Operators

Genetic operators are the heart of GAs, creating new solutions by combining and modifying parents. They must ensure every artist is scheduled exactly once, keeping all lineups valid [2]. The operators include selection (picking parents), crossover (combining parents), and mutation (making small changes), each designed to balance exploration and exploitation of new solutions by maintaining and prolifering building blocks in the population. All operators are implemented in **library/genetic_algorithm**.

2.4.1. Selection

Selection is probabilistic, based on fitness. It picks strong solutions but keeps enough variety to avoid early convergence. Two methods are tested:

- **Rank Selection** sorts all individuals by fitness, assigning the worst a rank of 1, the second-worst a rank of 2, and so on, with the best getting the highest rank. Parents are chosen via a roulette wheel with linear probabilities proportional to rank. This prevents top solutions from overpowering early, promoting diversity for better exploration. [2]
- **Tournament Selection** randomly picks k individuals (here, $k = 50$) and chooses the fittest as a parent. A larger k focuses on the best solutions (higher selection pressure), while a smaller k allows more exploration. It's widely used because it's faster (no need to evaluate all individuals) and easier to tune selection pressure.

2.4.2. Crossover

Crossover combines two parent lineups to create new ones, ensuring valid permutations where each artist appears exactly once. To apply crossover, the 2D schedule is flattened into a 1D list, operated on, and reshaped back to 2D. Two crossover operators, focused on avoiding *Position Problem of Standard Crossover*, are used with a probability (p_c) of 0.9 consistent to common boundaries ($p_c \in [0.8, 1]$) of GAs literature.

- **Cycle Crossover (CX)** identifies cycles of artist positions between parents (P), copying entire cycles alternately to offspring (O). Odd cycles assign P1's positions to O1 and P2's to O2; even cycles reverse this, as shown in **Figure F1**. This preserves exact artist positions, potentially important for N_{pop} objective. [2]
- **Order Crossover (OX)** takes a chunk of artist IDs from one parent and copies it to the child. The remaining slots are filled with artists from the other parent, keeping their order. If the child already has an artist being inserted, it skips to the next. [4] This preserves blocks of artists and their relative sequence, which can help maintain good stage or slot arrangements. See **Figure F2** for an example.

2.4.3. Mutations

Mutation randomly modifies a portion of the syntactic structure of an existing lineup solution to add variety and escape suboptimal solutions, maintaining valid permutations of the entire solution. Three mutation operators are applied with a probability (p_m) of 0.2, a common value in GA literature ($p_m \in [0, 0.2]$) to effectively balance exploration and exploitation. [2][4] Keeping p_m constant across configurations ensures a fair comparison of operator effects, isolating their impact on performance.

- **Prime Slot Swap Mutation** (**Figure F3**) swaps an artist in a prime slot (last one) with another from a non-prime slot. This directly targets prime slot popularity, aiming to place high-popularity artists in key slots.
- **Insert Mutation** (**Figure F4**) removes an artist from one position and inserts it into another, shifting others to fill the gap. This can significantly modify all fitness objectives, making it the most disruptive operator.
- **Stage Shuffle Mutation** (**Figure F5**) randomly shuffles all artists within one stage, keeping the stage's structure intact. This makes changes only on one stage's lineup, but it impacts all fitness criteria.

All chosen mutations are designed to change both the solution (genotype) and its fitness (phenotype), ensuring that any permutation has an impact on the fitness. Exceptions to this only occur in rare cases, such as when the involved artists have identical values, the sum of the objective values is the same, or in similar situations.

3. EXPERIMENTAL RESULTS AND DISCUSSION

The GAs works by mimicking evolution (**Figure B1**). It starts with a population of 100 random, valid lineups (each a 2D array of artist IDs). Over 100 generations, parents are chosen using Rank or Tournament Selection. With a **90%** chance (p_c), parents undergo crossover (OX or CX); otherwise, they're replicated. Offspring then face mutation (Prime Slot Swap, Insert, or Stage Shuffle) with a **20%** chance (p_m). The new population, determined by generational choice where offsprings automatically become the next generation, replaces the old one, and if elitism is enabled, the best lineup is carried over to avoid losing the top solution.

To find the best GA setup, a grid search (coded in *4. Genetic Algorithm Execution* section in notebook) tested 24 configurations: 2 selections (Rank, Tournament) × 2 crossovers (OX, CX) × 3 mutations (Prime Slot Swap, Insert, Stage Shuffle) × 2 elitism settings (True, False). Each configuration was run 30 times to account for randomness, with results evaluated using Median Best Fitness (MBF), robust to outliers, and standard deviation to measure stability and performance variability across generations.

3.1. Genetic Algorithm Performance

Observing **Figure H1** and **Figure H2** for all 24 configurations, a group of six notably poor performers consistently utilized **Rank Selection** and **OX**, regardless of mutation or elitism. Among these, configurations without elitism exhibited lower and unstable fitness curves with higher standard deviations and fluctuations, unlike their elitist counterparts. The highest fitness overall (0.878) (**Figure H9**) came from configuration with Rank selection, CX, Stage Shuffle mutation, and without Elitism, but the same configuration, when maintaining the best individual from the previous population, achieved the highest MBF (0.857), which can reinforce elitism's consistency. It is important to note that, despite this analysis, a higher or lower number of generations could lead to a different best configuration, given the significant performance similarity among the top five configurations.

For the remaining 18 configurations, visual inspection was inconclusive, necessitating statistical tests. To facilitate this analysis, we focused on the top five configurations (**Figure H3**) exhibiting the highest final MBF. All five employed Rank Selection and CX with an MBF > 0.853 and tight standard deviations $(0.007 - 0.011)$. A **Kruskal-Wallis** test (**Annex A**, $p - value = 1.0$) on these configurations found no significant median differences due to their similar high performance, fixing a significance level of 5% ($\alpha = 0.05$). The ten **Pairwise Mann-Whitney U** tests ($p - values: 0.13 - 0.92$) confirmed this, indicating that none of these five configurations was statistically superior to the others within this top group.

To understand the isolated impact of the chosen genetic operators, we employed an **Aggregate Runs Approach**. This involves grouping all experimental runs that share a specific hyperparameter value (e.g., all runs using *"Rank Selection"*) and then computing the MBF trend across these aggregated runs. This method allows us to evaluate an operator's general performance beyond its success in just one top configuration.

While **Rank Selection** was part of some of the highest MBF-scoring configurations, it also contributed to the worst-performing ones, consequently in **Figure H4**, it exhibited significantly higher variability in its results, with a standard deviation approximately three times greater than that of **Tournament Selection**.[2] Although the difference in the final MBF between the two methods was not vast, the latter demonstrated more consistent performance and a slightly superior overall median trend. Furthermore, if selection pressure were to be fine-tuned, the performance gap might widen, as Tournament Selection's pressure is more easily adjusted via a single parameter (tournament size).

In our runs, **CX** significantly outperforms **OX** balancing exploration and exploitation (**Figure H5**). OX converged faster, thereby exploring less of the search space. In our problem, preserving the exact artist position — which

[2] **Rank Selection** evaluates all individuals in the population to assign ranks, whereas **Tournament Selection** evaluates only a subset of individuals per tournament, making it computationally lighter. For a more rigorous analysis, results should be calculated and plotted by fitness evaluations rather than generations to account for differing computational effort.

CX does more effectively — appears more relevant than preserving relative sequence. This is critical for calculating popularity and, indirectly, the other fitness components, whereas the specific artist performing immediately before or after another has no direct impact on the fitness function.

Mutations show similar medians, with **Stage Shuffle** slightly better due to broader changes aiding escape from local optima (**Figure H6**). Even though low mutation probability ($p_m = 0.2$) limits their impact, **Insert**'s disruptions also enhance exploration, while **Prime Slot Swap** refines popularity, leading to a slightly faster convergence.

Lastly, **Figure H7** shows all configurations slightly prioritize popularity and diversity, reaching near-optimal values by generation 30, while conflict reduction lags due to its complexity, despite equal objective weights.

3.2. Comparative Analysis with Other Metaheuristics

The GA configuration with the best MBF considered above was compared to HC and SA, each with $10{,}000$ fitness evaluations ($= 100$ population $\times$ 100 generations) for fairness. HC, using intra-stage artist swaps (**Figure G1**), achieved a low MBF (0.744 ± 0.038), often trapped in local optima due to its greedy nature. SA, with similar swaps (**Figure G2**) and parameters $\boldsymbol{C}$: 100.0, $\boldsymbol{L}$: 10, $\boldsymbol{H}$: 1.5, reached an MBF of 0.858 (± 0.010).

Figure H8's boxplot compares final fitness, showing HC's lower median and wider spread compared to GA and SA. A **Kruskal-Wallis test** (**Annex A**) ($p - value = 0.00$, fixing $\alpha = 0.05$) confirmed significant differences among the three methods, given our chosen hyperparameters. **Dunn's post-hoc test** further revealed that both GA and SA significantly outperform HC ($p - value < \alpha$), but GA and SA show no significant difference when applying **Mann-Whitney U test** ($p - value = 0.595$). SA had a slightly higher median, but GA had a greater consistency, demonstrated by the absence of outliers and reliable identification of high-quality solutions.

It is worth noting that this analysis is valid only for the hyperparameters used, as no further tuning was performed. While additional tuning could have provided deeper insights into the algorithms' performance, it was not the main focus of our work. Nevertheless, based on the current results, both GA and SA have demonstrated to be competent algorithms to solve our problem.

4. CONCLUSION

This study highlights the effectiveness of GAs for optimizing music festival lineups, balancing the goals of maximizing prime slot popularity, ensuring genre diversity, and minimizing fan conflicts. Among 24 tested configurations, Rank Selection paired with Cycle Crossover delivered the best performance, with mutations and elitism providing subtle yet stabilizing benefits. This top GA configuration equalled SA in efficacy while significantly surpassing HC. While analysing each operator separately can help us understand their individual impacts, the best final solution comes from looking at the hyperparameters as a whole.

Given our results, the fitness values approached the theoretical maximum of 1, but the observed slowdown in improvement suggests possible premature convergence that limited further improvements. Increasing diversity—through higher mutation rates, fitness sharing, or larger population sizes—could mitigate this issue and enhance optimization outcomes.

Future work should explore expanding the parameter search space, including a higher maximum number of generations and tunning crossover, p_c, and mutation probabilities, p_m. Additionally, implementing Pareto-based multi-objective GAs could yield a diverse set of trade-off solutions, as multi-objective optimization typically produces multiple optimal solutions rather than a single one, enhancing the exploration of our composite fitness function's uniqueness.

In summary, our GA surpasses HC and rivals SA with the tested settings. Its full potential could be further unlocked by exploring additional hyperparameter configurations. By systematically tuning generational limits, operator rates, and embracing multi-objective frameworks, future studies can further enhance solution diversity, robustness, and the interpretability of fitness landscapes in the complex domain of festival scheduling.

BIBLIOGRAPHICAL REFERENCES

[1] Siarry, P. (Ed.) (2016). Metaheuristics, Springer.

[2] Vanneschi, L., & Silva, S. (2023). Lectures on Intelligent Systems. Springer Nature.

[3] Cicirello, V. (2023). A Survey and Analysis of Evolutionary Operators for Permutations. *Proceedings of the 15th International Joint Conference on Computational Intelligence*, 288–299. https://doi.org/10.5220/0012204900003595

[4] Katoch, S., Chauhan, S. S., & Kumar, V. (2020). A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, *80*(5). https://doi.org/10.1007/s11042-020-10139-6

[5] Ławrynowicz, A. (2011). Genetic Algorithms for Solving Scheduling Problems in Manufacturing Systems. *Foundations of Management*, *3*(2), 7–26. https://doi.org/10.2478/v10238-012-0039-2

[6] El-Ghazali Talbi, & Muntean, T. (2002). *Hill-climbing, simulated annealing and genetic algorithms: a comparative study and application to the mapping problem*. https://doi.org/10.1109/hicss.1993.284069

[7] Kruskal, W. H., & Wallis, W. A. (1952). Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, *47*(260), 583–621. https://doi.org/10.1080/01621459.1952.10483441

[8] DataTab. (2024). *Kruskal-Wallis-Test*. Datatab.net. https://datatab.net/tutorial/kruskal-wallis-test

[9] Dunnett, C. W. (1955). A Multiple Comparison Procedure for Comparing Several Treatments with a Control. *Journal of the American Statistical Association*, *50*(272), 1096–1121. https://doi.org/10.1080/01621459.1955.10501294

[10] Mann, H. B., & Whitney, D. R. (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, *18*(1), 50–60. https://doi.org/10.1214/aoms/1177730491

[11] DATAtab team. (2023). *Mann-Whitney U-Test*. Datatab.net. https://datatab.net/tutorial/mann-whitney-u-test

APPENDIX A. LITERATURE REVIEW

Table A1 – Literature Review.
(Chronologically ordered from the most recent article to the oldest)

Paper/Article/Book Title	Abstract Summary	Methodology	Main Findings	Crossover Methods Used	Mutation Methods Used	Selection Method Used	Reference
A Survey and Analysis of Evolutionary Operators for Permutations	Surveys and empirically analyses evolutionary operators (crossover and mutation) for permutations. It evaluates which permutation features (e.g., element position, edges, precedence) these operators effectively optimize using artificial fitness landscapes. All operators were implemented in the Chips-n-Salsa open-source Java library.	Empirical comparison using an adaptive Evolutionary Algorithm (EA) on the "Permutation in a Haystack" problem. Five artificial fitness landscapes were defined using different distance metrics to isolate specific permutation features. Performance was measured by the average solution cost (distance to target permutation) over 100 runs, compared against a baseline using only Swap mutation. Permutation length n=100.	CX, PMX, UPMX, and PBX optimize element positions; EER, ER, and OX effective for undirected edges, OX excels on directed edges. CX, NWOX, UOBX, OX2, PPX, UPPX, PMX, UPMX, and PBX effective for precedences; several also effective for cyclic precedences. Mapping of operators to optimized features provided.	CX, ER, EER, OX, NWOX, UOBX, OX2, PPX, UPPX, PMX, UPMX, PBX (all empirically compared). HeurX, EAX were also mentioned.	Swap, Adjacent Swap, Insertion (Jump), Reversal (Inversion), 2-change, 3opt, Block-Move, Block-Swap, Cycle (Cycle(kmax), Cycle(α)), Scramble (Shuffle), Uniform Scramble, Rotation, and Windowed.	Binary Tournament Selection.	[3]

A review on genetic algorithm: past, present, and future	Discusses recent GA advances, covering operators (crossover, mutation, selection, encoding), variants (single/multi-objective, parallel, chaotic, hybrid, real/binary coded), pros/cons, and applications. Aims to guide researchers with future directions.	Systematic review following PRISMA guidelines. Searched Google Scholar and PubMed using keywords (e.g., "Genetic Algorithm", "operators of GA"). Applied multi-stage filtering (date: 2007-2020, duplicates, title/abstract/full-text relevance) resulting in 30 selected papers.	**Crossover:** Single/k-point crossover simple but less diverse; uniform crossover good for large subsets; order and cycle crossovers improve exploration; PMX offers better performance; reduced and cycle crossovers risk premature convergence. **Mutation:** DM/SIM easy but risk premature convergence; scramble affects large gene segments, may degrade quality. **Selection:** Roulette (simple, risk premature convergence), rank (preserves diversity, slow), tournament (diversity-preserving, parallelizable, risk with large k), Boltzmann (finds global optima, slow), SUS (fast, unbiased), and elitism (preserves best, risks diversity loss).	Single-point, K-point, Uniform, Partially Matched (PMX), Order (OX), Precedence Preserving (PPX), Shuffle, Reduced Surrogate (RCX), Cycle (CX). (Also mentions Arithmetic, Blend, SBX, Laplace, Unimodal for Real-coded GAs)	Displacement (DM), Simple Inversion (SIM), Scramble (SM), Bit Flipping, Reversing. (Also mentions Exchange/Insertion as DM variants; Boundary, Power, Exponential, Polynomial for Real-coded GAs)	Roulette Wheel, Rank, Tournament, Boltzmann, Stochastic Universal Sampling (SUS), Elitism	[4]

Genetic Algorithms for Solving Scheduling Problems in Manufacturing Systems	Surveys recent GA advancements for advanced manufacturing scheduling, covering components like representation and operators across diverse shop types (parallel machines, flow shop, job shop, flexible job shop, open job shop, multi-factory). Proposes a Modified Genetic Algorithm (MGA) for distributed scheduling in industrial clusters, emphasizing chromosome encoding and operator use.	Literature review of GA components—encoding schemes (operation-based, job-based, random keys), crossover, mutation, and selection operators—and their use in manufacturing scheduling problems. Proposes a Modified Genetic Algorithm (MGA) for distributed scheduling in industrial clusters, featuring a novel two-step chromosome encoding (Type A and B for manufacturing and transport) and tailored operator selection.	GAs widely applied to scheduling with critical role of representation. Common operators: PMX and OX (crossover), inversion and insertion (mutation), roulette and tournament (selection). Proposed multi-level GA uses specialized encoding for jobs, operations, machines, factories, and transport orders to improve distributed scheduling in industrial clusters. Emphasizes coding challenges as key.	PMX, OX, CX, Position-Based Crossover.	Inversion mutation, Insertion mutation	Roulette Wheel Selection, Tournament Selection.	[5]
Hill-Climbing, Simulated Annealing and Genetic Algorithms: a comparative study	Compares HC, SA, and GA for combinatorial optimization problems.	Evaluates performance on TSP and 8-queens problems. HC uses greedy improvements, SA accepts worse solutions probabilistically, GA employs population-based evolution.	Simulated annealing and genetic algorithms better escape local optima than hill climbing. GA achieves higher solution quality but requires more computational resources.	Single-point crossover	Bit-flip mutation	Local (neighborhood-based) selection	[6]

APPENDIX B. GENETIC ALGORITHM FLOWCHART

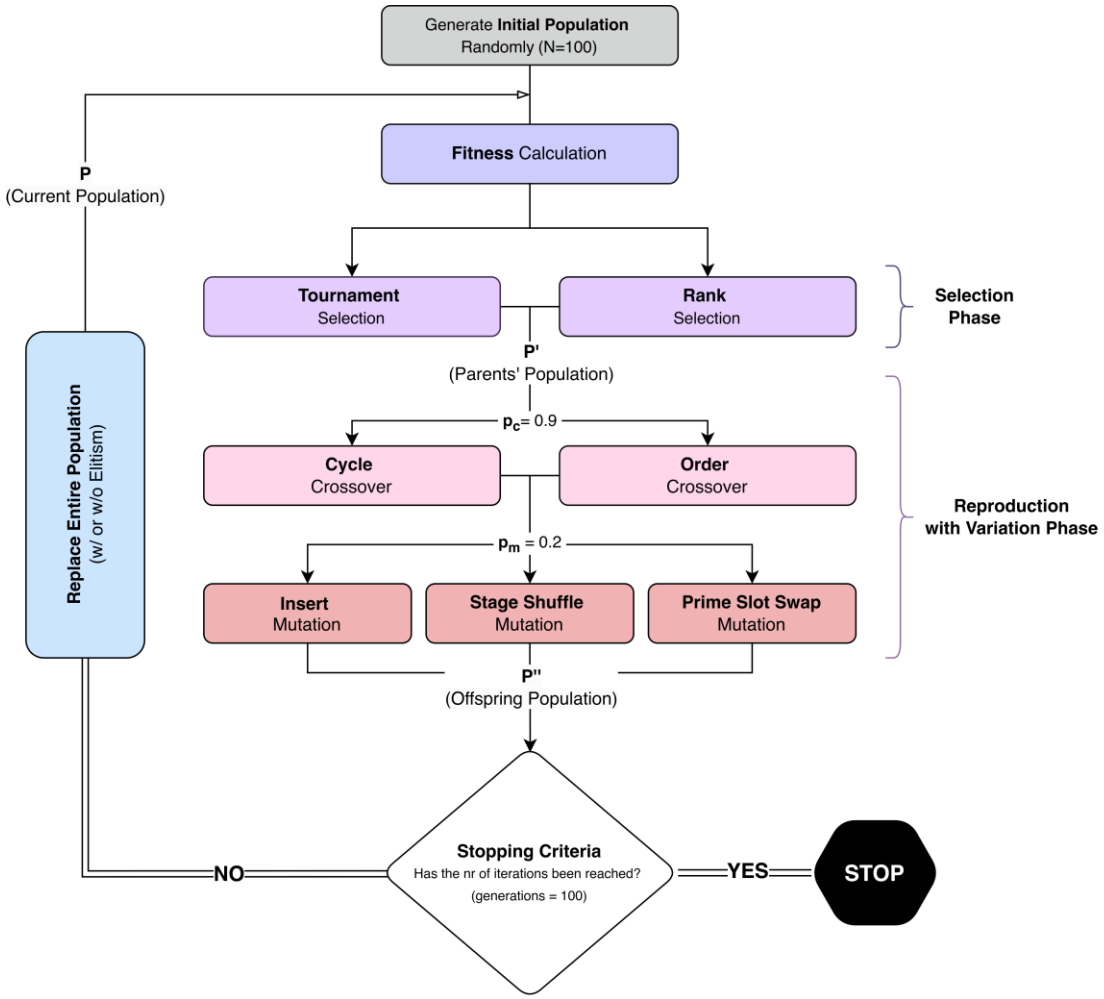


Figure B1 - Genetic Algorithm Flowchart.

APPENDIX C. REPRESENTATION OF THE SOLUTION

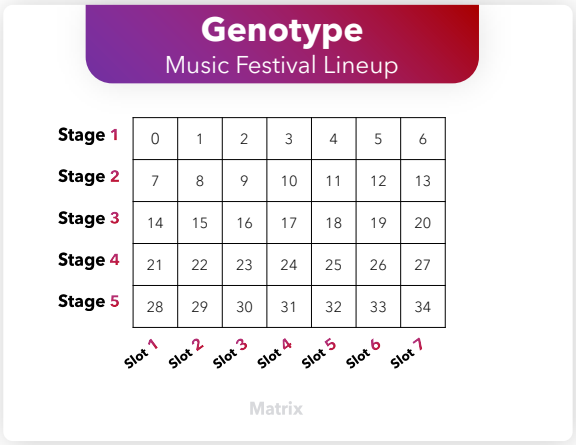


Figure C1– Solution Representation (*Genotype*).

APPENDIX D. CODE IMPLEMENTATION

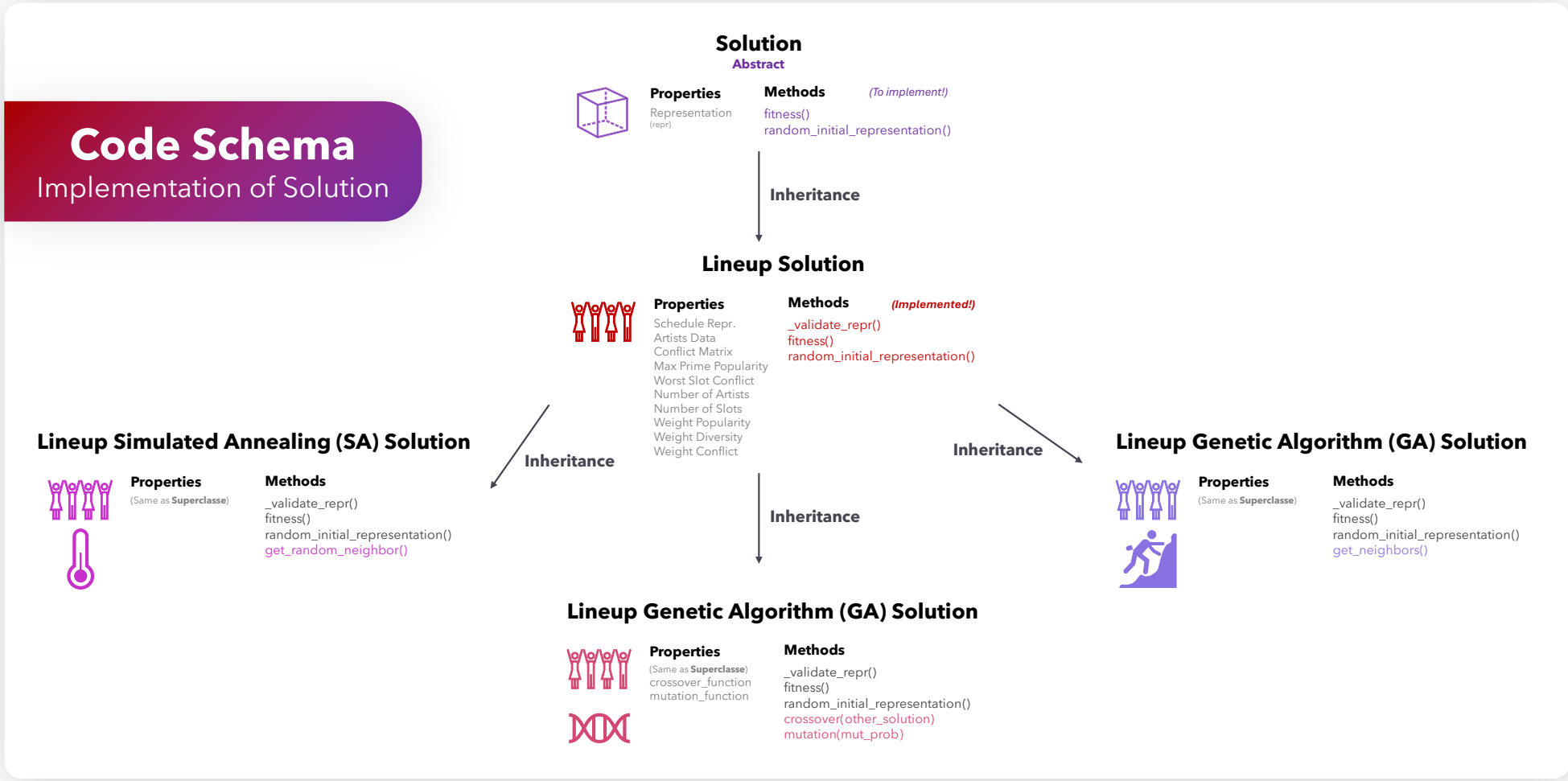


Figure D1 – Code Schema Implementation.

APPENDIX E. FITNESS FUNCTION CALCULATION EXAMPLE

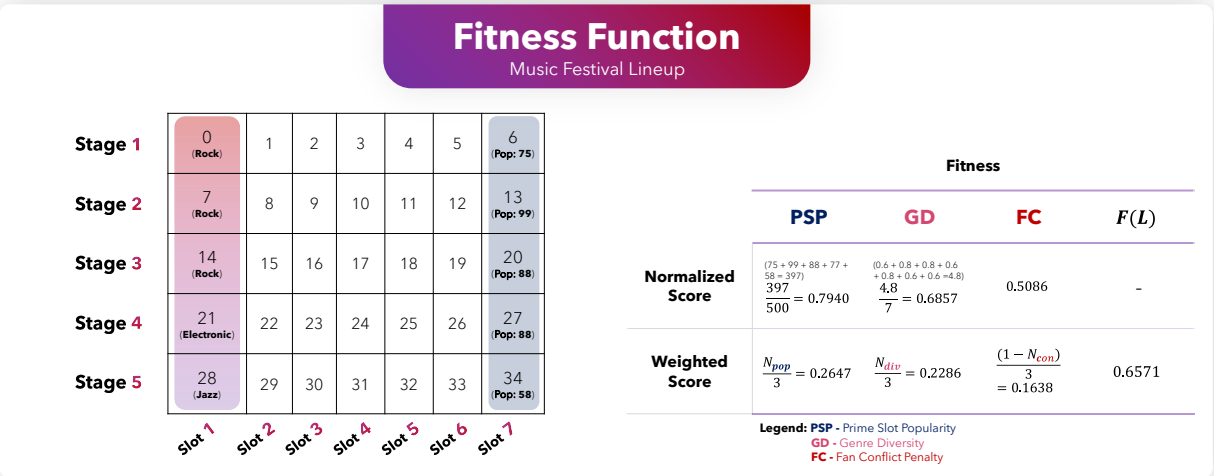


Figure E1 – Illustration of Fitness Calculation (Phenotype) with an example.

To further illustrate the practical application of the fitness function described in **Section 2.3**, this appendix provides a step-by-step calculation for an exemplary festival lineup. The visual representation of this calculation (**Figure E1**), includes the specific artists' popularity and the derivation of normalized scores.

- **Normalized Prime Slot Popularity (N_{pop}):**
  - The artists scheduled in the prime slots (last time slot of each stage) are **[6, 13, 20, 27, 34]**.
  - Their combined popularity is $75 + 99 + 88 + 77 + 58 = 397$.
  - Normalized against a maximum possible prime slot popularity of **500**, this yields:

$$N_{pop}(L) = \frac{75 + 99 + 88 + 77 + 58}{100 \times 5} = \frac{397}{500} = 0.7940$$

- **Normalized Genre Diversity (N_{div}):**
  - The diversity is calculated for each of the 7 time slots by determining the ratio of unique genres present to the maximum possible unique genres (**5** in this case).

$$N_{div}(L) = \frac{\frac{3}{5} + \frac{4}{5} + \frac{4}{5} + \frac{3}{5} + \frac{4}{5} + \frac{3}{5} + \frac{3}{5}}{7} = \frac{0.6 + 0.8 + 0.8 + 0.6 + 0.8 + 0.6 + 0.6}{7} = \frac{4.8}{7} \approx 0.6857$$

- **Normalized Fan Conflict Penalty (N_{con}):**
  - For each time slot, conflicts between artists on different stages are summed and normalized against a worst-case slot conflict of **10** ($\boldsymbol{C_2^{10} \times 1}$ – Corresponds to all possible combinations between artists in the same slot and considering the maximum theoretical conflict – **1**)

$$N_{con}(L) = \frac{1}{7}\left(\frac{6.8}{10} + \frac{2.4}{10} + \frac{5.35}{10} + \frac{7.35}{10} + \frac{3.0}{10} + \frac{5.4}{10} + \frac{5.3}{10}\right) = \frac{3.56}{7} \approx .0.5086$$

For example: $\text{Slot 0 Conflict} = c_{0,7} + c_{0,14} + c_{0,21} + c_{0,28} + c_{7,14} + c_{7,21} + c_{7,28} + c_{14,21} + c_{14,28} + c_{21,28} =$
$= 1.0 + 1.0 + 0.0 + 1.0 + 1.0 + 1.0 + 0.65 + 0.65 + 0.5 + 0.0 =$
$= 6.8$

Overall Fitness Calculation for the Example:

Using the normalized component scores derived above:

$$F(L) \approx \frac{1}{3} \times 0.7940 + \frac{1}{3} \times 0.6857 + \frac{1}{3} \times (1 - 0.5086)$$

$$F(L) \approx \frac{1}{3} \times 0.7940 + \frac{1}{3} \times 0.6857 + \frac{1}{3} \times 0.4914$$

$$F(L) \approx 0.2647 + 0.2286 + 0.1638$$

$$F(L) \approx 0.6571$$

Therefore, the overall fitness for the illustrative lineup presented in **Figure E1** is approximately **0.6571**.

APPENDIX F. GENETIC ALGORITHM OPERATORS

Crossovers

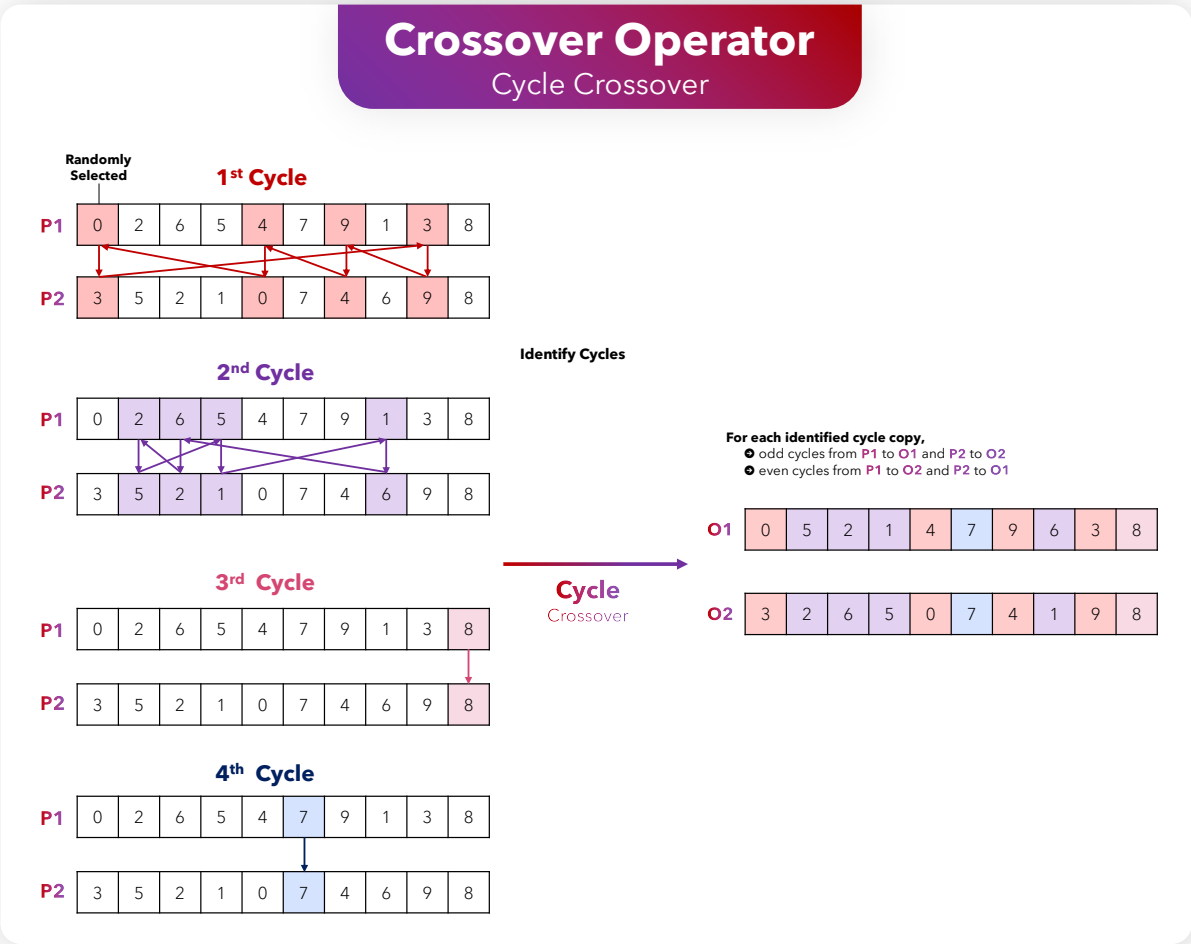


Figure F1 – Cycle Crossover (CX).

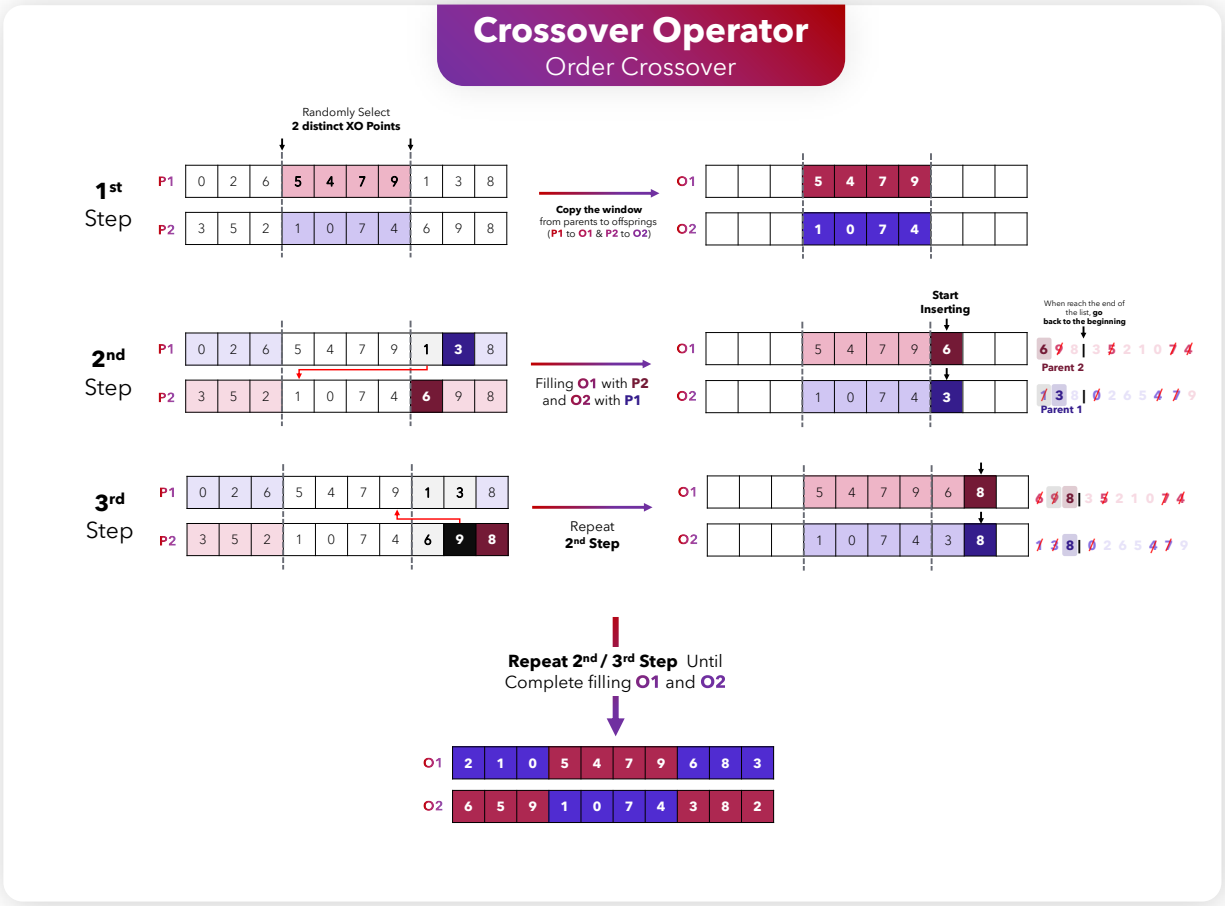


Figure F2 – Order Crossover (OX).

Mutations

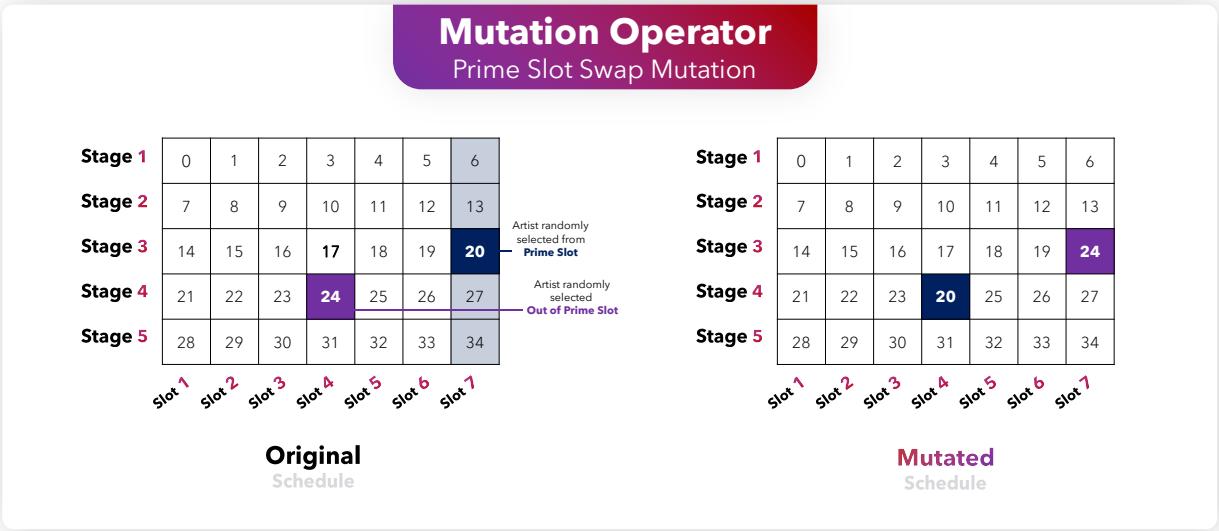


Figure F3 – Prime Slot Swap Mutation.

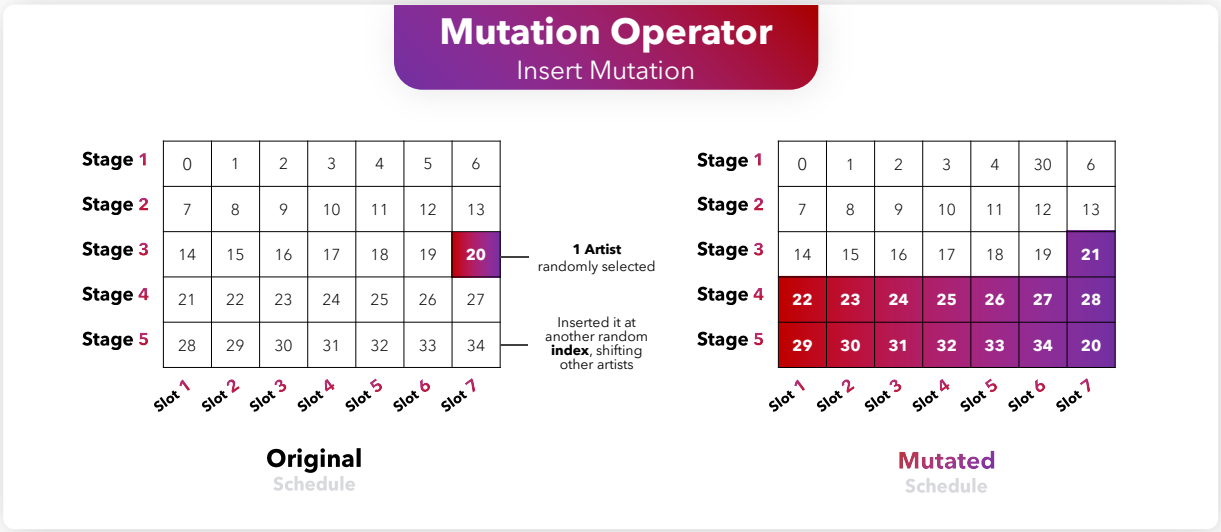


Figure F4 – Insert Mutation.

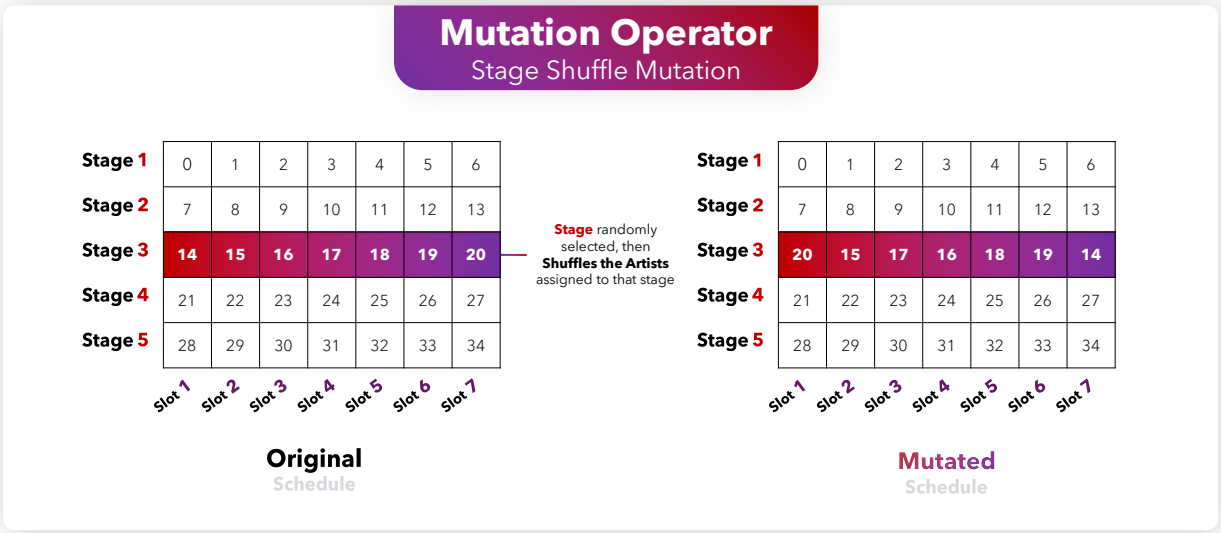


Figure F5 – Stage Shuffle Mutation.

APPENDIX G. SA & HC NEIGHBOURS

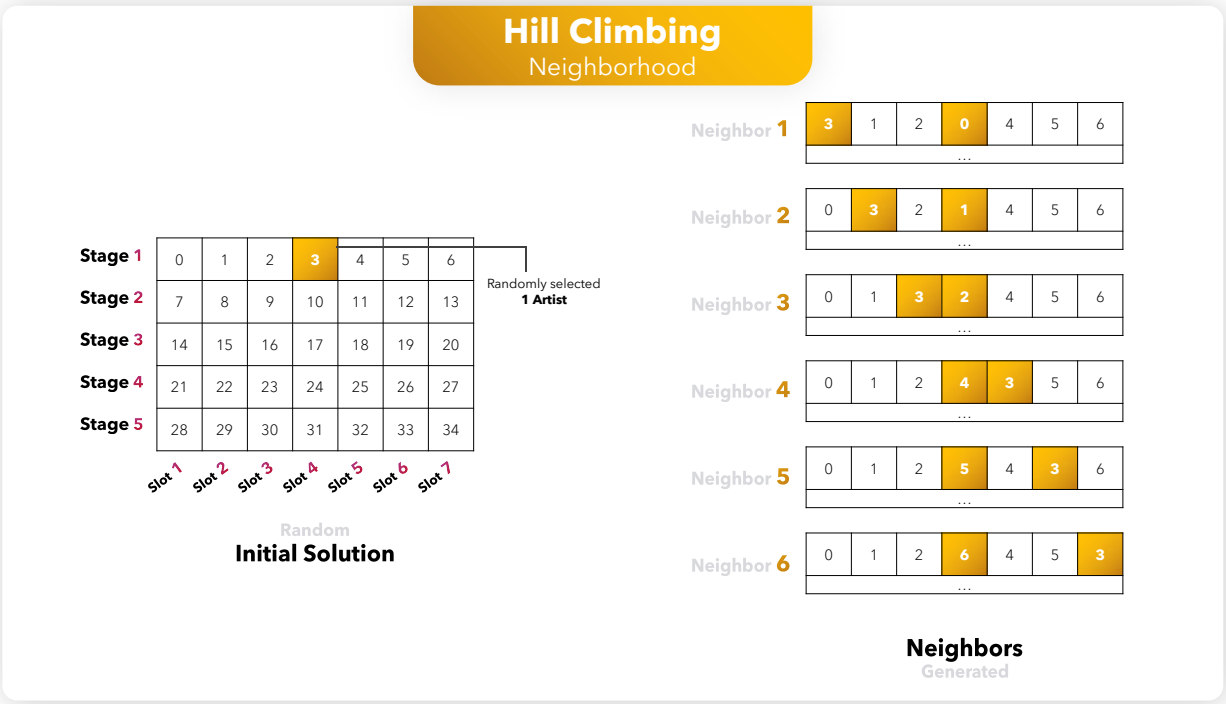


Figure G1 – Illustration of the generation of a Neighborhood in Hill Climbing.

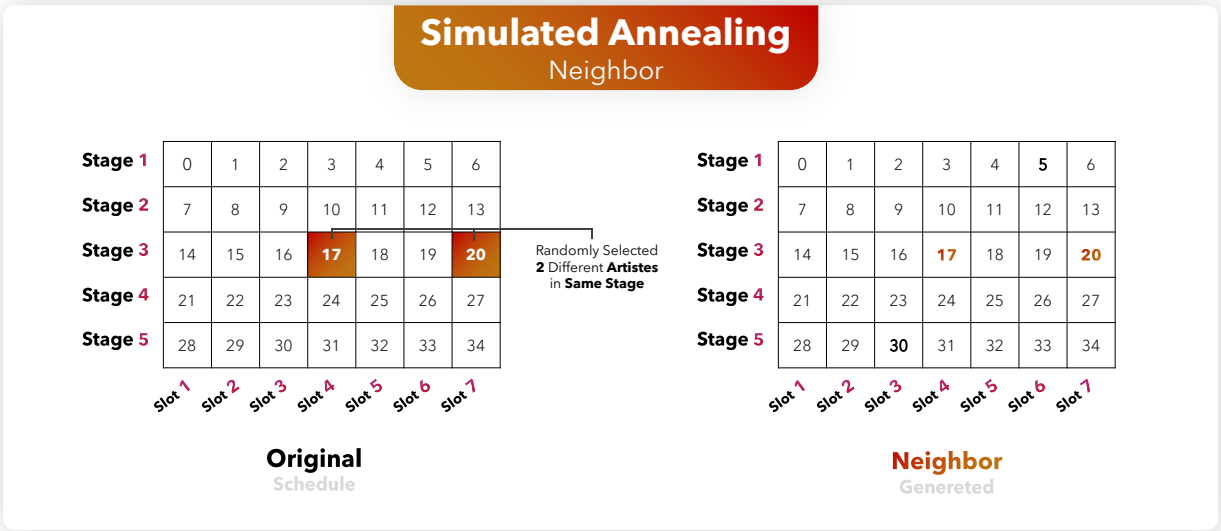


Figure G2 – Illustration of the generation of a Neighbor in Simulated Annealing.

APPENDIX H. RESULTS OF GRIDSEARCH

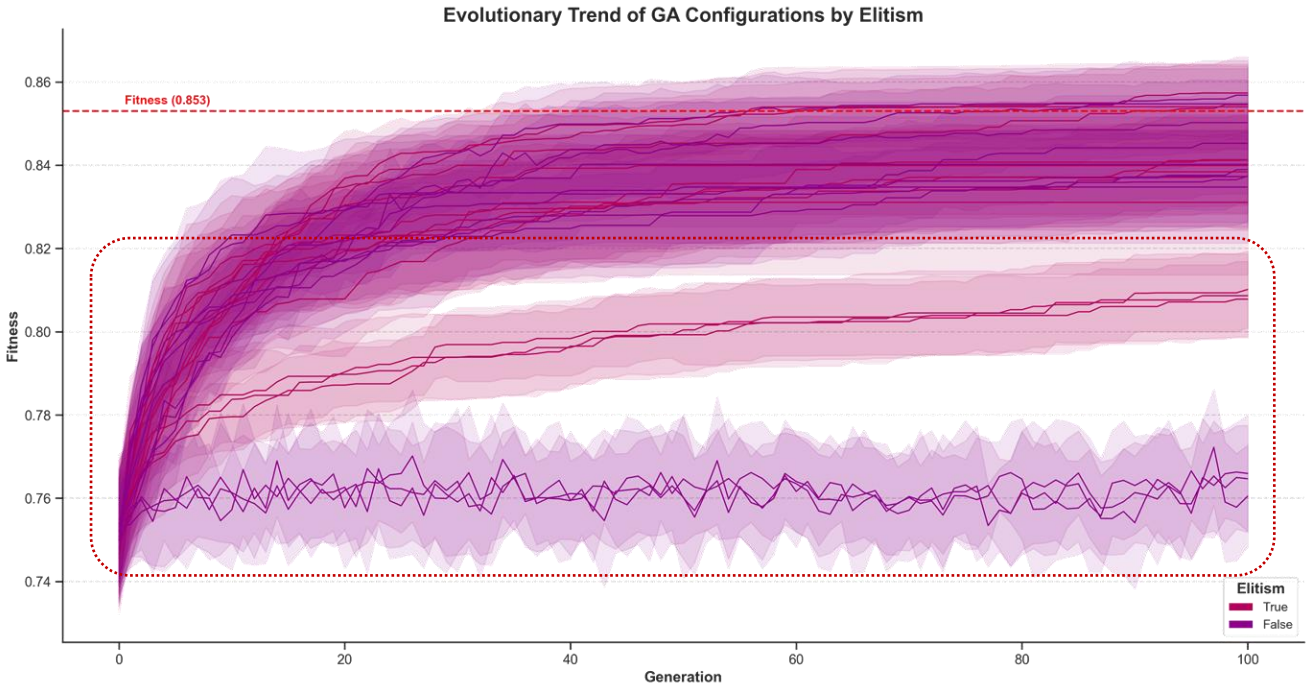


Figure H1 – Evolutionary trend of all GA configurations, showing median fitness (± standard deviation) over 100 generations.

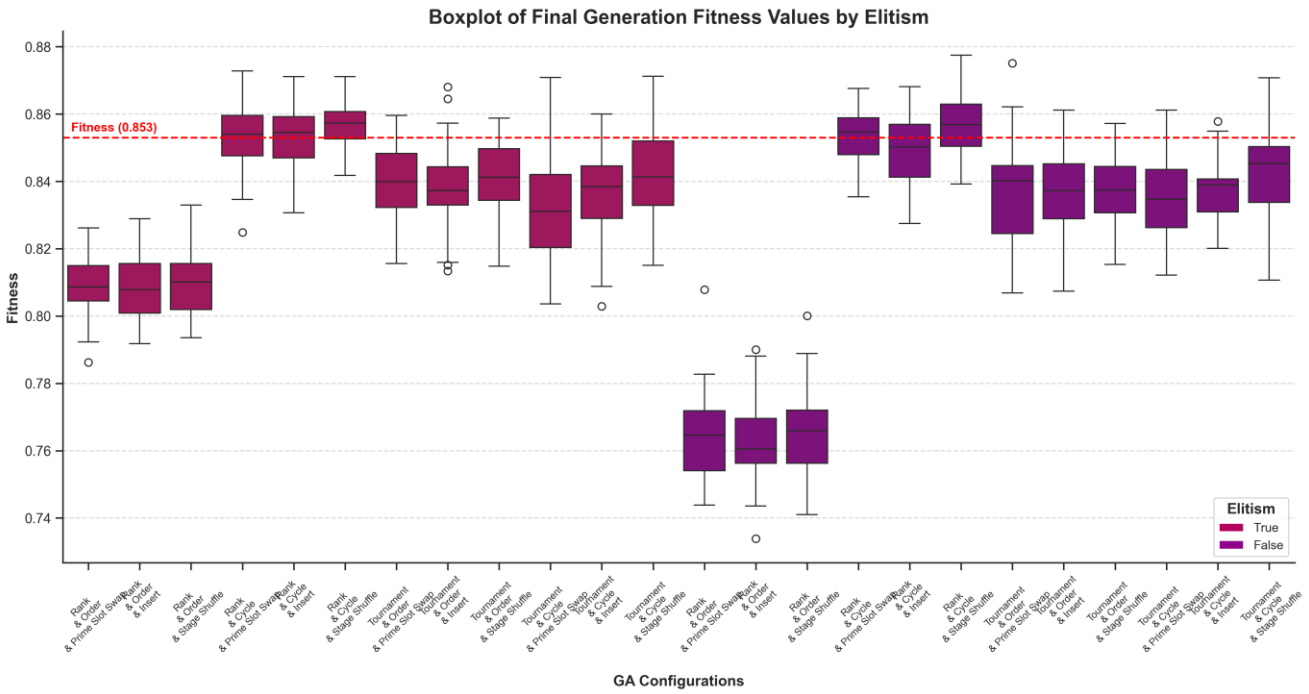


Figure H2 – Boxplot of final generation fitness's for all 24 configurations, coloured by elitism status (**True:** pink, **False:** purple). A **red** dashed line at 0.853 marks a reference fitness for top 5 configurations.

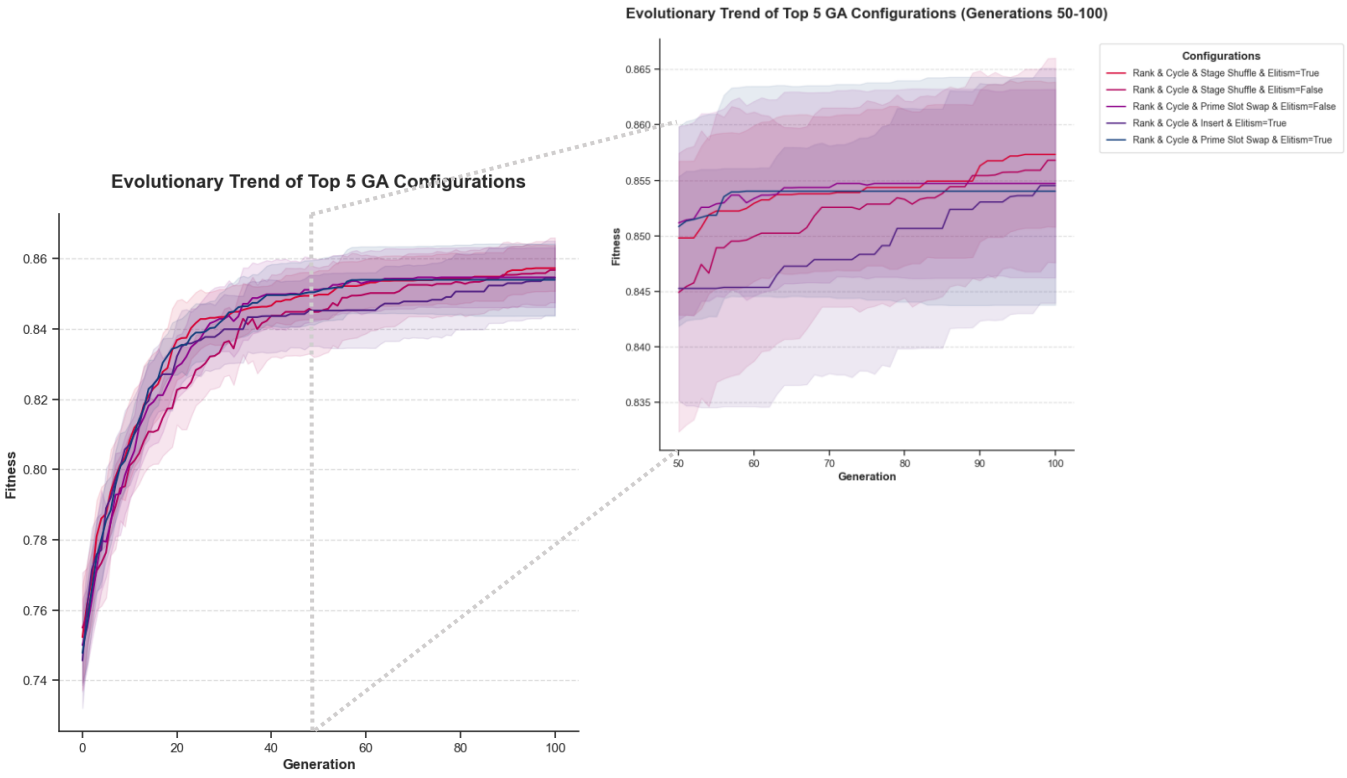


Figure H3 – Evolutionary trend of the top five GA configurations, showing median fitness (± standard deviation) over 100 generations. Zoomed-in evolutionary trend (generations 50–100) for the top five GA configurations.

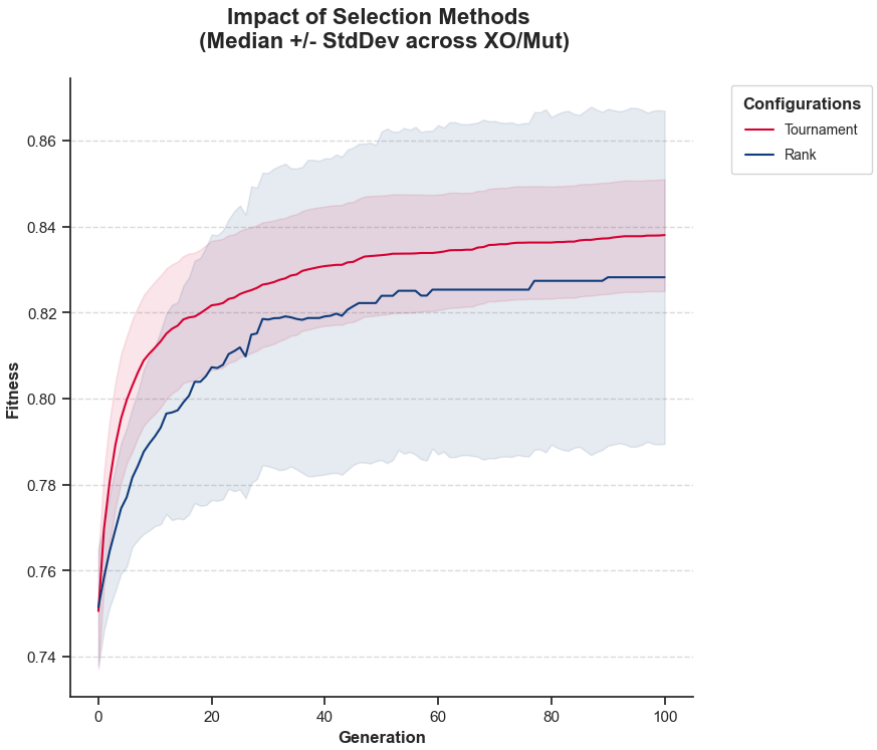


Figure H4 – Impact of selection methods (Rank vs. Tournament) on median fitness (± standard deviation) across all crossover and mutation combinations.

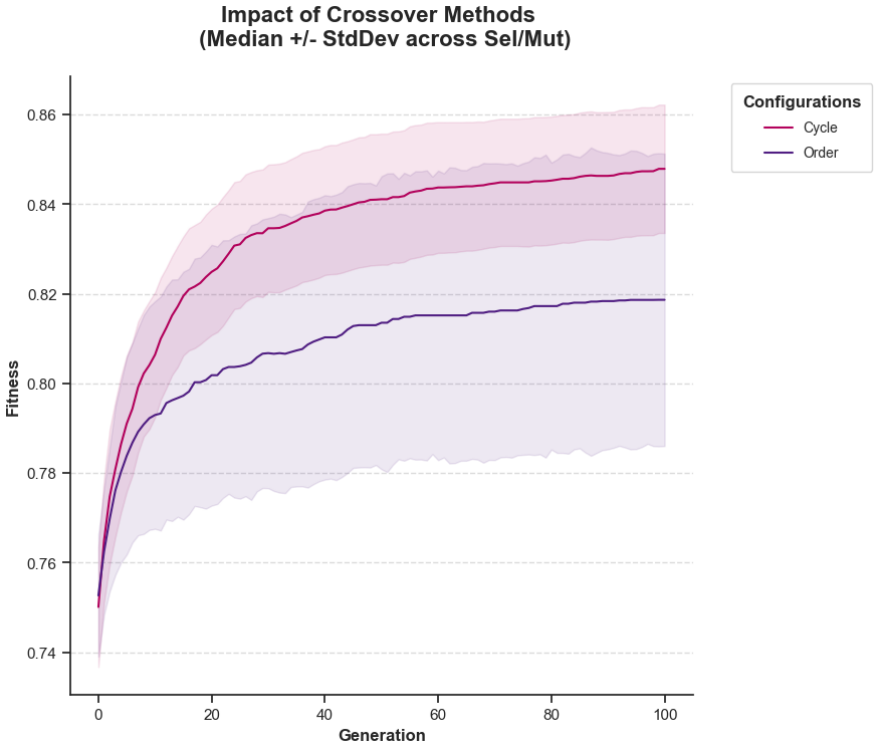


Figure H5 – Impact of crossover methods (Cycle vs. Order) on median fitness (± standard deviation) across all selection and mutation combinations.

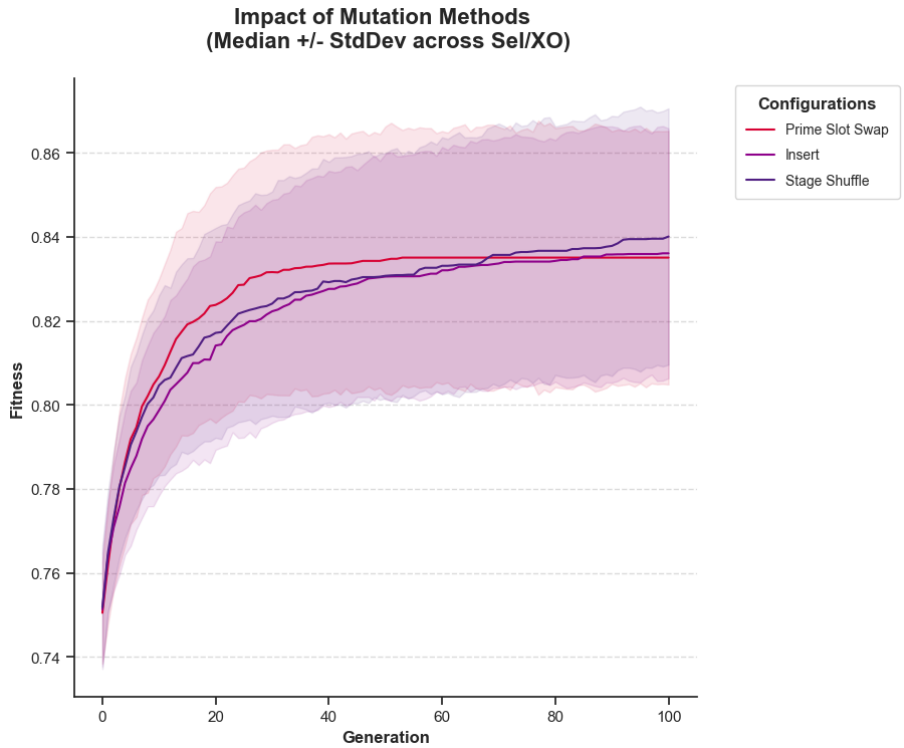


Figure H6 – Impact of mutation methods (Prime Slot Swap, Insert, Stage Shuffle) on median fitness (± standard deviation) across all selection and crossover combinations.

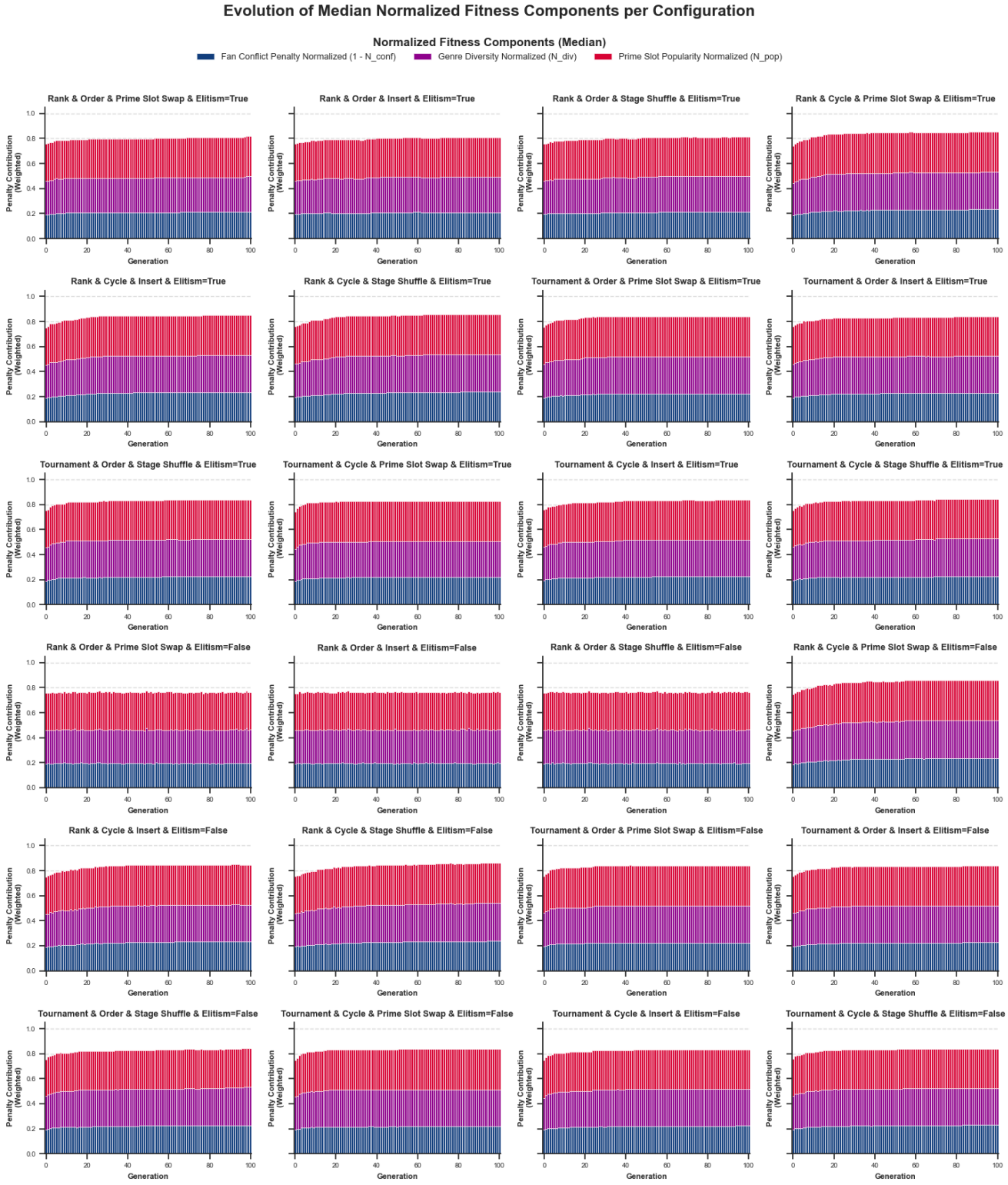


Figure H7 – Stacked bar charts for all 24 GA configurations, showing median penalty contributions (popularity, diversity, conflict) over generations.

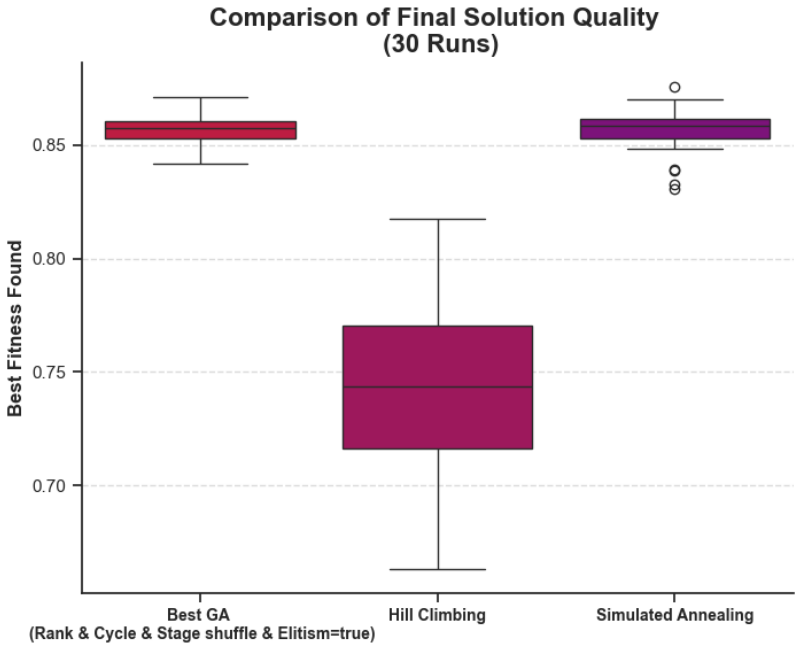


Figure H8 – Boxplot comparing final fitness's of the best GA configuration (*Rank & Cycle & Stage Shuffle & Elitism=True*), HC, and SA across 30 runs.

Figure H9 – Final Best Lineup Solution found in GAs.

ANNEX A. STATISTICAL TESTS FOR COMPARISON OF DIFFERENT CONFIGURATIONS OR ALGORITHMS

We employed non-parametric statistical tests, as the assumption of normal distribution is not guaranteed due to the independence of each run, impossibilitating the use of parametric tests like the t —test. Non-parametric tests rank data rather than relying on distributional assumptions. The **Kruskal–Wallis** [7] test extends the *Mann–Whitney U* to compare more than two groups by testing for differences in median rank distributions. If Kruskal–Wallis finds a significant effect, **Dunn's test** [9] performs pairwise post-hoc comparisons while controlling the family-wise error rate. In contrast, the **Mann–Whitney U** [10] is a two-sample test that directly compares rank sums between exactly two independent groups. Each method has distinct use cases, assumptions, and computational steps, making them complementary tools for robust comparison of algorithmic configurations.

Kruskal–Wallis Test [7][8]

The Kruskal–Wallis test is a non-parametric alternative to one-way ANOVA that evaluates whether three or more independent samples come from the same distribution of ranks. It assigns ranks to all observations pooled across groups and then compares the mean rank per group to detect shifts in central tendency. Under the null hypothesis, all groups share the same median rank; the test statistic H approximates a chi-square distribution with $k-1$ degrees of freedom (k = number of groups). Key assumptions are that samples are independent, observations are at least ordinal, and group distributions have similar shape (but not necessarily normality).

Procedure:

1. Pool all data and assign ranks (average ranks for ties).
2. Compute group sum of ranks R_i and sample size n_i.
3. Calculate

$$H = \frac{12}{N(N+1)} \sum_{i=1}^{k} \frac{R_i^2}{n_i} - 3(N+1),$$

   where N = total observations.
4. Compare H to χ^2_{k-1}; reject null if H is large.

Dunn's Post-Hoc Test [9][8]

Dunn's test performs pairwise comparisons of mean rank differences following a significant Kruskal–Wallis result, controlling for multiple testing.

It calculates a z-statistic for each pair:

$$Z_{ij} = \frac{\bar{R}_i - \bar{R}_j}{\sqrt{\frac{N(N+1)}{12}\left(\frac{1}{n_i} + \frac{1}{n_j}\right)}},$$

where $\bar{R}_i$ and $\bar{R}_j$ are mean ranks for groups i and j.

P-values from these z-tests are then adjusted by Bonferroni, Holm, or other procedures to maintain the overall α level. Unlike naïve pairwise Mann–Whitney tests, Dunn's method retains the pooled variance structure implied by the omnibus test.

Mann–Whitney U Test [10][11]

The Mann–Whitney U (or Wilcoxon rank-sum) test compares two independent groups by evaluating differences in rank sums.

It tests the null hypothesis that observations in one sample tend to be equally likely to be higher or lower than those in the other sample.

The U-statistic is given by

$$U = n_1 n_2 + \frac{n_1(n_1 + 1)}{2} - R_1$$

where R_1 is the sum of ranks in sample 1 and n_1, n_2 are sample sizes.

When both $n_1, n_2 > 20$, U approximately follows a normal distribution, enabling z-approximation for p-values.

Assumptions include independent observations and ordinal or continuous data; the test does not require normality but assumes similar shapes of distributions.

Comparison of Tests

Feature	Kruskal–Wallis [7]	Dunn's Test [9]	Mann–Whitney U [10]
Purpose	Omnibus comparison of ≥ 3 groups	Pairwise post-hoc after Kruskal–Wallis	Direct two-group comparison
Data	≥ 3 independent samples, ordinal/continuous	Same as Kruskal, for pairs	2 independent samples, ordinal/continuous
Assumptions	Independence, ordinal data, similar shapes	Inherits from Kruskal–Wallis	Independence, ordinal data, similar shapes
Test Statistic	$\boldsymbol{H} \sim \chi^2$	$\boldsymbol{Z_{ij}}$ for each pair	U or z –approximation
Multiplicity	Single omnibus test	P-value adjustment (Bonferroni, Holm, etc.)	Typically, single test; adjust if multiple