Solving the Rubik's Cube with Genetic Algorithms

Inês Araújo (20240532)
Jorge Cordeiro (20240594)
Rita Palma (20240661)
Tomás Silva (20230982)

NOVA IMS, NOVA University of Lisbon, Portugal

May 2025

Abstract

This study presents a comprehensive genetic-algorithm approach to solving the 3×3 Rubik's Cube, whose approximately 4.30×10^{19} reachable configurations pose an immense search challenge. We extend prior work by developing a flexible cube representation that supports all eighteen face and slice moves and by implementing interchangeable crossover, mutation and selection operators with automatic elimination of redundant moves.

After the evolutionary phase, candidate solutions undergo a multigene local search followed by a greedy look-ahead refinement. We systematically evaluate every combination of mutation, crossover and selection strategies via grid search, then employ random search to tune hyperparameters including population size, chromosome length, crossover and mutation probabilities, elitism proportion and generation limit.

Experiments comprising thirty independent runs for each of twelve operator schemes identify an optimal configuration—200 individuals, 30-move chromosomes, 60 % crossover probability, 30 % mutation probability, 1 % elitism, segment mutation, two-point crossover and tournament selection over 1500 generations. Wilcoxon signed-rank tests confirm that both refinement phases yield significant performance gains. Although only two trials achieved perfect solutions for ten-move scrambles, the algorithm consistently reduces scramble disorder, underscoring the promise of genetic algorithms as domain-agnostic solvers for complex combinatorial challenges.

The notebook is available at `https://github.com/tomasestrocio/CIFO_PROJECT`.

Contents

1 Introduction

The Rubik's Cube is one of the best-known combinatorial puzzles, boasting approximately 4.30×10^{19} reachable states. Because an optimal solution requires searching an astronomically large space, the puzzle is an appealing benchmark for evolutionary computation. It is worth noting that Several deterministic algorithms can efficiently solve the Rubik's Cube, including Kociemba's Two-Phase Algorithm [2] and the CFOP method used by speedcubers [3].

Despite these established techniques, exploring genetic algorithms for this problem offers unique research value. GAs provide an alternative approach that doesn't rely on predetermined sequences or exhaustive search, potentially yielding new insights about both the puzzle and evolutionary computation. Our work examines whether properly configured genetic algorithms can successfully navigate this complex state space to find valid solutions. This report presents a complete pipeline that couples a *Genetic Algorithm* (GA) with two intensification heuristics (local multi-gene search and greedy look-ahead) to obtain high-quality solutions from scrambled configurations of ten random moves.

Our implementation draws conceptual inspiration from Saeidi [1], yet extends that work in five directions:

1. A flexible cube model and move system capable of representing all possible cube states and manipulations.
2. Multiple interchangeable genetic operators, including different types of mutation, crossover, and selection, combined with automatic cleaning of redundant moves.
3. Two additional improvement steps after the genetic algorithm: a local search that modifies small parts of the solution, and a greedy search that applies X best moves until no further progress is possible.
4. A complete grid search to compare all possible combinations of mutation, crossover, and selection strategies, identifying the four best combinations based on experimental results.
5. A random search for fine-tuning the genetic algorithm parameters based on the four best operator combinations.

2 Methodology

2.1 Rubik's Cube Representation and Fitness Measure

State encoding. The cube is stored as a Python dictionary whose keys are the six faces—`U`, `R`, `F`, `D`, `L`, `B`. Each face is a 3×3 *colour matrix* whose entries are integers from $\{1, \ldots, 6\}$. A single coloured square on the physical puzzle is called a **facelet**; thus every matrix contains one *centre facelet* (the immutable reference colour of that face) and eight *non-central facelets*. The solved configuration is the unique state in which all nine facelets on every face share the same integer label. Our implementation provides a comprehensive Cube class with methods for initialization, copying, move application, and fitness evaluation. See Table 1 in Appendix A for a description of the cube structure and movements, which may help with the code interpretation.

Move set. Eighteen elementary moves are supported: the twelve quarter-turn rotations of the six faces $(U, U', R, R', \ldots, B, B')$ plus the six middle-slice rotations M, M', E, E', S, S'. Each move updates both the local 3×3 matrix (via `np.rot90`) and the corresponding edge/column vectors on the neighbouring faces, reproducing standard speed-cubing notation.

Fitness function. Let S_f denote the set of the eight non-central facelets on face $f \in \{U, R, F, D, L, B\}$ and let $\mathrm{col}(s)$ return the integer colour of facelet s. The fitness of a cube state C is the total number of mis-coloured facelets:

$$F(C) = \sum_{f \in \{U,R,F,D,L,B\}} |\{s \in S_f \mid \mathrm{col}(s) \neq \mathrm{centre}(f)\}|$$

Because each of the six faces contributes exactly eight non-central facelets, the score satisfies

$$0 \ \leq \ F(C) \ \leq \ 48,$$

with $F = 0$ for a solved cube and $F = 48$ when every inspected facelet disagrees with its face centre.

Expected score of a random cube. A *uniformly random state* of the cube can be modeled by independently assigning to every non-central facelet one of the six colours with equal probability. Let X_k be the random variable associated with the state of the k-th non-central facelet:

$$X_k = \begin{cases} 1 & \text{if facelet } k \text{ mismatches its face centre,} \\ 0 & \text{otherwise,} \end{cases}$$

Then $F = \sum_{k=1}^{48} X_k$. Since $\Pr[X_k = 1] = 5/6$ and the X_k are i.i.d.,

$$\mathbb{E}[F] = \sum_{k=1}^{48} \mathbb{E}[X_k] = 48 \times \frac{5}{6} = 40.$$

Thus a uniformly random cube is expected to score 40, i.e. about 83% of the inspected facelets are in the wrong position on average.

2.2 The Genetic Algorithm

The solution search relies on a standard Genetic Algorithm (GA) whose key components are tailored to the Rubik's Cube domain yet remain algorithm-agnostic enough for systematic hyper-parameter exploration. Our GeneticAlgorithm class implements all core evolutionary operators with configurable strategies (see Table 2 in Apendix A for comprehensive method documentation).

Chromosome representation. An individual is a variable-length integer vector $\mathbf{m} = (m_1, m_2, \dots, m_\ell)$ where each gene $m_i \in \{1, \dots, 18\}$ encodes one of the eighteen legal cube moves (12 face turns plus 6 slice turns). Unless otherwise stated, the default chromosome length is $\ell = 30$ moves, but this value can be tuned during the experiments.

Population initialisation. The initial population of size N is generated by sampling genes independently and uniformly over $\{1, \dots, 18\}$ while forbidding immediate inverse pairs (e.g. R followed by R'). In a adiction, we implemented a function `clean_chromosome` which removes any newly introduced inverse pairs after crossover or mutation, keeping solutions concise.

Genetic operators.

- **Mutation**. Three variants are implemented: *simple*—each gene is replaced with a new random move with probability p_{mut}; *swap*—two distinct positions are chosen uniformly and their moves are exchanged; *segment*—a pair of cut points $(i < j)$ is sampled and the subsequence $m_i, \dots, m_j$ is reversed.
- **Crossover**. Two parents recombine, with probability p_{cross} , by either*one-point* or *two-point* crossover: *one-point* chooses a single cut position c and swaps the suffixes $(m_{c+1}, \dots, m_\ell)$; *two-point* uses two cut points $(c_1 < c_2)$ and exchanges the middle segment.
- **Selection**. Reproductive pressure is applied through either fitness-proportionate roulette wheel sampling or a deterministic $k = 3$ tournament that returns the fittest competitor among three uniformly drawn candidates.

Adaptive mutation. To avoid premature convergence, the base mutation probability p_{mut} is automatically increased whenever the global best fitness fails to improve for 50 consecutive generations:

$$p_{\text{mut}} \leftarrow \min\bigl(1.2\, p_{\text{mut}},\ 1\bigr).$$

Elitism and stopping criterion. Let p_{elite} be the *elitism rate.* After fitness evaluation the best $\lceil p_{\text{elite}} N \rceil$ individuals are copied verbatim to the next generation, guaranteeing that no improvement is ever lost.

Evolution proceeds until either:

1. a chromosome achieves a perfect fitness of 0
2. the generation counter reaches the user-specified cap $G_{\max}$.

2.3 Solution Improvement After Genetic Algorithm

After the genetic algorithm finds its best solution, two additional improvement steps are applied to try to further reduce the fitness. Both steps call the same `clean_chromosome` routine used earlier, so that adjacent inverse moves are eliminated automatically (see Table 3 in Appendix A for the complete method overview).

Multi-gene local search. Starting from the best chromosome, the algorithm performs up to $I_{\text{LS}} = 50$ improvement sweeps. For every position i in the chromosome (and for every ordered pair (i, j) or triple (i, j, k)) a candidate sequence is built by re-sampling one, two, or three moves uniformly from $\{1, \ldots, 18\}$. The candidate replaces the original gene(s) and is accepted if it yields a fitness decrease. Whenever a modification is accepted the current chromosome and fitness are updated immediately.

Greedy look-ahead. The locally-optimised cube state then enters a greedy loop that explores all legal move sequences of length one, two, and three. At each iteration the single sequence that produces the largest instantaneous fitness drop is appended to the solution and applied to the cube. If no sequence yields progress the loop halts; otherwise it continues for at most $I_{\text{GR}} = 100$ iterations.

3 Results

3.1 Grid Search for Operator Evaluation

To assess the impact of different combinations of genetic operators, we performed a systematic grid search covering all possible combinations of mutation types (*simple*, *swap*, *segment*), crossover strategies (*one_point*, *two_point*), and selection methods (*roulette*, *tournament*), yielding 12 unique configurations. All experiments were conducted with fixed genetic algorithm parameters: Pop. = 100, Chrom. = 30, $p_{\text{cross}} = 0.8$, $p_{\text{mut}} = 0.2$, $p_{\text{elite}} = 0.10$, and $G_{\max} = 1500$.

The results of this grid search are presented in Table 4 (Appendix B). The table reports the mean final fitness $\bar{f}$ and the best observed fitness $f_{\min}$ over 30 independent runs for each operator combination.

3.2 Random-Search Hyper-parameter Tuning

After identifying the four best operator configurations through grid search, we proceeded with a second optimization stage to fine-tune the genetic algorithm's numerical parameters. This stage used random search to explore combinations of Pop., Chrom., p_{cross}, p_{mut}, p_{elite}, and $G_{\max}$.

The ranges of values considered for each parameter are presented in Table 5 (Appendix C). For each of the four selected operator combinations, we sampled 30 random configurations and evaluated each configuration over 30 independent runs.

The detailed results of this random search are presented in Appendix C, in Tables 6, 7, 8, and 9. Each table reports the tested parameter configurations along with the corresponding mean final fitness $\bar{f}$ and best observed fitness $f_{\min}$.

3.3 Final Assessment of the Best Performing Configuration

After comparing the results of all random-search experiments, the best configuration was identified as (Pop. = 200, Chrom. = 30, $p_{\text{cross}} = 0.6$, $p_{\text{mut}} = 0.3$, $p_{\text{elite}} = 0.01$, $G_{\max} = 1500$), using *segment* mutation, *two-point* crossover, and *tournament* selection.

To further explore the behaviour of this configuration, we performed 30 additional runs. The evolution of the mean fitness over generations is reported in Figure 1, and the distribution of final fitness scores across these runs is presented in Figure 2 in the Appendix D.

4 Discussion

Looking at the results from the grid search in Table 4 (Appendix B), we can identify several patterns regarding the effectiveness of different operator combinations.

Between the three mutation types tested, *simple* mutation leads to lower mean fitness compared with the other ones. Regarding the crossover operators, both *one-point* and *two-point* crossover have similar performances. There is no clear advantage of one over the other, meaning that the crossover choice may not be as important in this optimization problem. On the other hand, the selection strategy may play a more significant role in this situation. *Tournament* selection outperforms roulette in most combinations, especially when combined with *simple* and *segment* mutation, which is an indication that using *tournament* selection may be better suited for the problem at hand. The configurations that showed the best results based on the mean fitness values and the ones that we selected to do further optimization explorations were:

- *Simple* mutation, *one-point* crossover and *tournament* selection
- *Simple* mutation, *two-point* crossover and *tournament* selection
- *Segment* mutation, *one-point* crossover and *tournament* selection
- *Segment* mutation, *two-point* crossover and *tournament* selection

The experimental results yield several clear insights into how genetic algorithm parameters affect optimization performance.

Across all experiments, population sizes between 100 and 150 consistently yielded the best results, indicating a balance between diversity and efficiency.

For crossover probability, higher values, particularly $p_{\text{cross}} \geq 0.7$, led to better average fitness across all mutation–crossover combinations, emphasizing the importance of effective recombination.

Mutation probabilities around 0.1 to 0.2 resulted in the most successful runs, while a value of 0.3 often degraded performance, likely due to excessive randomness affecting the selection of advantageous traits. Elitism rates between 1% and 10% helped preserve strong individuals without causing premature convergence, contributing to more stable optimization. The combination of *segment* mutation with *one-point* crossover achieved the highest average fitness, indicating a positive interaction between these operators.

These results suggest that balanced parameters and well-chosen operators significantly improve optimization performance.

Finally, we wanted to asses the impact that the two improvements implemented, multi-gene local search and greed look-ahead, have on the final result. To do so, we tested our best configuration with and without these improvements. The results, illustrated in the boxplots in Figure 3 in Appendix E, indicate that the incorporation of these improvements lead to better performance.

To support this observation with statistical evidence, we conducted the Wilcoxon signed-rank test. The resulting p-value of 0.033 allows us to conclude that the difference in the two algorithms is statistically significant. Therefore, we can say with certainty that applying the improvements results in an enhancement to the algorithm's performance.

5 Conclusion and Future Work

This work presented a genetic algorithm for solving the 3x3 Rubik's Cube, employing different mutation, crossover, and selection operators. The main conclusions are:

- **Effectiveness of the Genetic Algorithm:** We demonstrated that genetic algorithms can effectively solve the Rubik's Cube, with multiple configurations achieving perfect solutions (fitness = 0).
- **Importance of Operator Selection:** The appropriate choice of genetic operators has a significant impact on performance, with certain combinations consiterforming others.

From the results, we also concluded that 30 runs may not provide a sufficiently robust performance variability estimate, potentially requiring a larger number of repetitions to ensure statistical reliability.

As future work, additional genetic operators could be tested, and a broader comparison with other optimization algorithms should be carried out.

References

[1] Saeidi, S. (2018). Solving the Rubik's Cube using Simulated Annealing and Genetic Algorithm. *International JourAppendix: of Education and Management Engineering, 8(1), 1–10* https://doi.org/10.5815/ijeme.2018.01.01

[2] Kociemba, H. (1992). Close to God's Algorithm. *Cubism For Fun*, 28, 10–13. https://www.kociemba.org/index.html

[3] Fridrich, J. (1997). Speed cubing - My system for solving the Rubik's Cube. http://www.ws.binghamton.edu/fridrich/system.html

A Apendix: Implementation details

Table 1: Cube class methods and move functions used for Rubik's cube representation

Cube Class Methods	
`__init__()`	Initializes a solved 3×3 cube with six faces (`U, R, F, D, L, B`), each as a 3×3 NumPy array with unique face IDs (1–6).
`copy()`	Returns a deep copy of the cube.
`rotate_face(face, clockwise)`	Rotates a given face (clockwise by default) without affecting others.
`apply_move(move)`	Applies a numbered move (1–18) using the corresponding move function.
`apply_sequence(sequence)`	Applies a sequence of moves in order.
`fitness()`	Returns the number of non-central facelets not matching the center facelet.
`print_cube()`	Prints the cube's current state to the console.
`init_moves()`	Static method that defines the 18 move functions: – Face turns: `U, U', D, D', F, F', B, B', L, L', R, R'` – Slice moves: `M, M', E, E', S, S'` Each move updates the rotated face and adjacent rows/columns.
Move Functions	
`move_U, move_Ui`	Rotate the upper face and update the top rows of `F, R, B, L`.
`move_D, move_Di`	Rotate the down face and update the bottom rows of `F, L, B, R`.
`move_F, move_Fi`	Rotate the front face and update the bottom of `U`, the left of `R`, top of `D`, and right of `L`.
`move_B, move_Bi`	Rotate the back face and update the top of `U`, right of `L`, bottom of `D`, and left of `R`.
`move_L, move_Li`	Rotate the left face and update the left of `U, F, D` and right of `B`.
`move_R, move_Ri`	Rotate the right face and update the right of `U, F, D` and left of `B`.
`move_M, move_Mi`	Rotate the vertical middle slice between `L` and `R`.
`move_E, move_Ei`	Rotate the horizontal middle slice between `U` and `D`.
`move_S, move_Si`	Rotate the front-back middle slice relative to the `F` face.

Table 2: Methods and attributes of the `GeneticAlgorithm` class

Constructor and Attributes	
`__init__()`	Initializes the algorithm with configuration parameters such as population size, chromosome length, mutation, crossover, and selection strategies.
`inverses`	Dictionary mapping each move to its inverse.
`mutation_type`	Mutation mode: `simple`, `swap`, `segment`, or `random`.
`crossover_type`	Type of crossover used: `one_point` or `two_point`.
`selection_type`	Selection strategy: `roulette` or `tournament`.
Chromosome Generation and Cleaning	
`is_inverse(move1, move2)`	Returns whether the two moves are inverse of each other.
`choose_random_move(prev)`	Chooses a random move avoiding the inverse of the previous one.
`generate_random_chromosome()`	Generates a chromosome of given length avoiding inverse pairs.
`clean_chromosome(chrom)`	Removes consecutive inverse moves from the chromosome.
`initial_population()`	Generates the initial population of chromosomes.
Fitness Evaluation and Selection	
`evaluate_population(pop)`	Computes fitness values for a population.
`roulette_wheel_selection(pop, fits)`	Selects a parent using fitness-proportional sampling.
`tournament_selection(pop, fits, k)`	Selects the best of k random individuals.
`select_parent(pop, fits)`	Applies the configured selection strategy to choose a parent.
Crossover Operators	
`one_point_crossover(p1, p2)`	Performs crossover at a random point.
`two_point_crossover(p1, p2)`	Performs crossover between two random points.
`crossover(p1, p2)`	Executes the selected crossover type.
Mutation Operators	
`mutate(chrom, rate)`	Mutates a chromosome using one of the following types: – `simple`: replace gene with a new random move – `swap`: exchange two gene positions – `segment`: reverse a subsequence of genes
Main Evolutionary Process	
`run()`	Executes the genetic algorithm loop with: – Initial population generation – Elitism, selection, crossover, and mutation – Adaptive mutation rate after stagnation – Termination when fitness = 0 or max generations is reached

Table 3: Methods and attributes of the `RubiksCubeSolver` class

Initialization and Scrambling	
`__init__(scramble_moves=10)`	Creates a scrambled cube using `scramble()`, storing the cube and scramble sequence.
`scramble(moves_count)`	Generates and applies a random sequence of moves to return a scrambled cube and the move list.
Main Solving Pipeline	
`solve()`	Main solver that executes the solving pipeline: – Genetic algorithm to get a base solution – Local search to refine it – Greedy improvement to finalize – Applies final sequence and shows fitness and state
Local and Greedy Improvements	
`local_improvement(chromosome, iterations=50)`	Applies iterative local changes to 1, 2, or 3 genes, selecting better candidates using fitness. Stops early if no improvement.
`greedy_improvement(cube, max_iters=100)`	Tries all 1-, 2-, and 3-move combinations to greedily improve cube state. Applies the best and repeats.
Visualization	
`plot_fitness(fitness_progress)`	Plots the best fitness per generation using Matplotlib.

B Appendix: Operator Evaluation Grid Search

Table 4: Operator screening over 30 trials per combination

Mutation	Crossover	Selection	Mean fitness	Best
simple	one_point	roulette	17.47	6.00
simple	one_point	tournament	15.53	0.00
simple	two_point	roulette	17.20	0.00
simple	two_point	tournament	15.67	4.00
swap	one_point	roulette	18.30	0.00
swap	one_point	tournament	19.87	6.00
swap	two_point	roulette	18.43	0.00
swap	two_point	tournament	18.43	0.00
segment	one_point	roulette	19.90	7.00
segment	one_point	tournament	17.03	4.00
segment	two_point	roulette	19.70	10.00
segment	two_point	tournament	16.67	0.00

C Appendix: Hyper-Parameter Random Search

Table 5: Hyper-parameter search ranges

Parameter	Values sampled
Population size (Pop)	50, 100, 150, 200, 300
Chromosome length (Chrom.)	30, 40, 50, 60
Crossover rate (p_{cross})	0.50, 0.60, 0.70, 0.80
Mutation rate (p_{mut})	0.10, 0.20, 0.30
Elitism rate (p_{elite})	0.01, 0.10
Generation cap (G_{max})	1000, 1500

Table 6: Simple mutation, One-point crossover, Tournament selection

Pop.	Chrom.	p_{cross}	p_{mut}	p_{elite}	G_{max}	$\bar{f}$	$f_{\min}$
100	50	0.8	0.2	0.01	1000	16.37	0.00
50	60	0.8	0.2	0.10	1000	22.47	10.00
50	40	0.7	0.2	0.10	1000	22.37	16.00
150	30	0.8	0.2	0.01	1500	15.77	0.00
200	60	0.7	0.2	0.10	1500	20.17	3.00
300	30	0.8	0.1	0.10	1000	19.00	2.00
150	40	0.8	0.1	0.01	1000	16.17	4.00
100	30	0.6	0.2	0.10	1500	16.90	0.00
300	50	0.6	0.2	0.01	1500	15.87	0.00
200	50	0.7	0.2	0.01	1500	16.33	2.00
50	30	0.7	0.2	0.10	1500	18.87	9.00
200	60	0.8	0.2	0.10	1500	17.07	0.00
200	50	0.6	0.2	0.01	1000	15.43	0.00
150	50	0.5	0.1	0.01	1000	17.83	7.00
200	60	0.8	0.3	0.10	1500	15.83	0.00
150	30	0.8	0.3	0.01	1500	16.93	3.00
300	30	0.7	0.2	0.10	1000	13.93	0.00
200	40	0.8	0.1	0.10	1000	22.03	15.00
200	50	0.5	0.3	0.10	1000	16.97	6.00
200	30	0.6	0.3	0.10	1000	16.03	0.00
300	30	0.6	0.2	0.10	1000	14.03	0.00
50	50	0.5	0.3	0.01	1500	15.67	0.00
50	40	0.6	0.3	0.10	1000	21.20	8.00
300	40	0.5	0.2	0.10	1500	15.27	4.00
50	50	0.8	0.1	0.10	1500	23.50	17.00
200	30	0.5	0.2	0.10	1000	16.33	0.00
100	30	0.8	0.1	0.01	1500	14.63	0.00
50	40	0.6	0.1	0.01	1500	16.17	3.00
150	50	0.7	0.2	0.10	1500	18.33	0.00
50	30	0.8	0.2	0.10	1000	19.30	4.00

Table 7: Simple mutation, Two-point crossover, Tournament selection

Pop.	Chrom.	p_{cross}	p_{mut}	p_{elite}	G_{max}	$\bar{f}$	$f_{\min}$
300	40	0.8	0.3	0.1	1000	14.80	0.00
100	50	0.6	0.2	0.01	1500	15.90	4.00
200	50	0.8	0.1	0.01	1000	16.63	7.00
100	50	0.8	0.1	0.1	1500	23.00	12.00
200	30	0.6	0.3	0.1	1000	14.93	2.00
300	40	0.7	0.2	0.01	1000	16.63	6.00
50	40	0.7	0.1	0.01	1000	15.80	0.00
150	50	0.6	0.1	0.01	1000	16.80	0.00
300	50	0.7	0.2	0.01	1000	16.90	0.00
150	60	0.8	0.2	0.1	1000	21.73	6.00
200	50	0.6	0.3	0.1	1500	15.20	0.00
300	50	0.8	0.1	0.1	1500	21.30	14.00
50	30	0.7	0.2	0.01	1000	17.37	4.00
200	30	0.6	0.3	0.1	1500	16.23	0.00
200	30	0.7	0.1	0.1	1500	18.13	4.00
200	50	0.5	0.1	0.01	1000	16.83	3.00
200	60	0.5	0.2	0.1	1000	20.63	0.00
150	60	0.5	0.3	0.1	1500	19.30	3.00
200	60	0.8	0.2	0.01	1500	16.13	0.00
50	30	0.7	0.3	0.01	1000	18.83	0.00
200	50	0.7	0.1	0.1	1000	22.90	17.00
150	50	0.6	0.1	0.1	1000	22.47	16.00
100	40	0.5	0.3	0.1	1500	16.50	0.00
150	60	0.7	0.2	0.01	1500	17.17	0.00
300	60	0.8	0.2	0.01	1500	18.27	0.00
200	30	0.5	0.2	0.1	1000	15.37	0.00
300	50	0.7	0.2	0.1	1000	18.07	2.00
50	30	0.7	0.3	0.1	1000	18.33	7.00
150	30	0.6	0.2	0.01	1000	18.07	0.00
100	60	0.5	0.3	0.01	1000	18.63	6.00

Table 8: Segement mutation, One-point crossover, Tournament selection

Pop.	Chrom.	p_{cross}	p_{mut}	p_{elite}	G_{max}	$\bar{f}$	$f_{\min}$
150	50	0.7	0.3	0.011	1000	17.23	2.00
50	60	0.8	0.3	0.1	1500	22.93	20.00
200	30	0.6	0.3	0.1	1000	18.63	0.00
100	50	0.6	0.1	0.1	1000	22.13	15.00
100	30	0.8	0.1	0.01	1500	17.97	7.00
150	30	0.8	0.3	0.01	1000	16.50	0.00
200	60	0.5	0.1	0.1	1500	21.87	18.00
150	30	0.5	0.1	0.01	1500	17.57	3.00
150	30	0.7	0.3	0.01	1500	18.33	8.00
150	60	0.5	0.2	0.01	1500	16.57	60.00
300	50	0.7	0.1	0.1	1000	21.63	17.00
50	50	0.5	0.2	0.1	1000	23.17	19.00
150	40	0.5	0.1	0.1	1500	22.03	18.00
300	30	0.6	0.3	0.01	1000	16.23	0.00
100	40	0.7	0.1	0.1	1000	21.60	13.00
100	40	0.5	0.2	0.01	1000	18.67	8.00
150	50	0.8	0.3	0.01	1000	18.20	0.00
150	60	0.6	0.1	0.1	1000	18.13	4.00
300	60	0.8	0.1	0.01	1000	17.77	3.00
50	50	0.7	0.2	0.1	1000	22.30	16.00
200	30	0.8	0.3	0.1	1000	19.53	12.00
300	50	0.7	0.1	0.1	1500	21.13	10.00
300	40	0.7	0.3	0.1	1500	20.50	7.00
100	30	0.8	0.1	0.01	1000	18.60	8.00
50	50	0.5	0.1	0.1	1500	22.33	18.00
150	30	0.6	0.3	0.01	1500	17.37	4.00
200	60	0.7	0.2	0.01	1500	17.33	4.00
300	60	0.8	0.1	0.1	1500	21.23	16.00
50	40	0.7	0.3	0.1	1000	21.87	15.00
100	30	0.5	0.2	0.1	1000	19.74	3.00

Table 9: Segement mutation, Two-point crossover, Tournament selection

Pop.	Chrom.	p_{cross}	p_{mut}	p_{elite}	G_{max}	$\bar{f}$	$f_{\min}$
300	30	0.7	0.1	0.1	1500	17.10	4.00
300	30	0.6	0.1	0.01	1500	17.40	0.00
300	30	0.8	0.2	0.01	1500	17.17	4.00
100	40	0.5	0.3	0.01	1000	18.23	4.00
50	50	0.6	0.1	0.01	1000	19.10	4.00
50	30	0.6	0.1	0.1	1500	20.73	11.00
150	60	0.7	0.2	0.1	1000	21.57	18.00
150	60	0.8	0.1	0.01	1000	14.87	0.00
200	30	0.6	0.1	0.01	1000	19.10	10.00
200	30	0.5	0.1	0.01	1500	15.53	4.00
150	50	0.8	0.2	0.01	1000	16.27	0.00
100	40	0.6	0.1	0.1	1500	21.03	6.00
150	30	0.5	0.1	0.01	1000	17.80	0.00
150	50	0.8	0.1	0.1	1000	21.67	16.00
150	40	0.8	0.1	0.1	1500	20.80	12.00
50	40	0.5	0.3	0.01	1500	17.50	0.00
300	50	0.6	0.3	0.1	1000	21.17	15.00
300	40	0.8	0.3	0.1	1000	19.37	6.00
50	40	0.6	0.3	0.1	1000	22.10	6.00
300	60	0.7	0.1	0.1	1500	21.50	17.00
100	40	0.6	0.3	0.1	1500	21.60	18.00
300	60	0.8	0.1	0.1	1500	21.33	17.00
200	50	0.5	0.2	0.01	1500	17.07	0.00
200	30	0.6	0.3	0.01	1500	13.87	0.00
200	30	0.8	0.1	0.1	1500	18.17	5.00
150	60	0.7	0.2	0.01	1500	17.40	4.00
200	30	0.7	0.2	0.1	1500	17.30	4.00
200	50	0.8	0.1	0.1	1000	21.63	18.00
300	60	0.7	0.1	0.1	1000	21.60	15.00
50	60	0.5	0.1	0.01	1000	18.03	4.00

D Appendix: Best Configuration Run Results

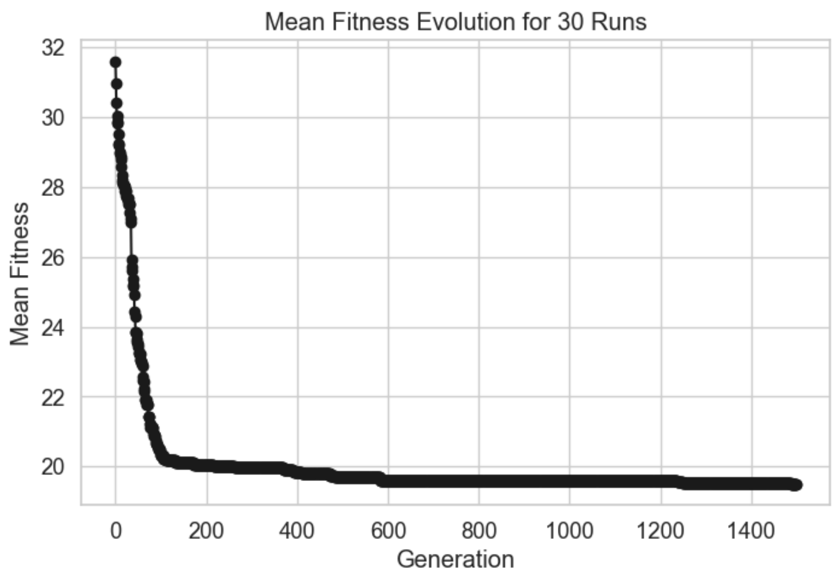

Figure 1: Mean fitness evolution over 30 independent runs.

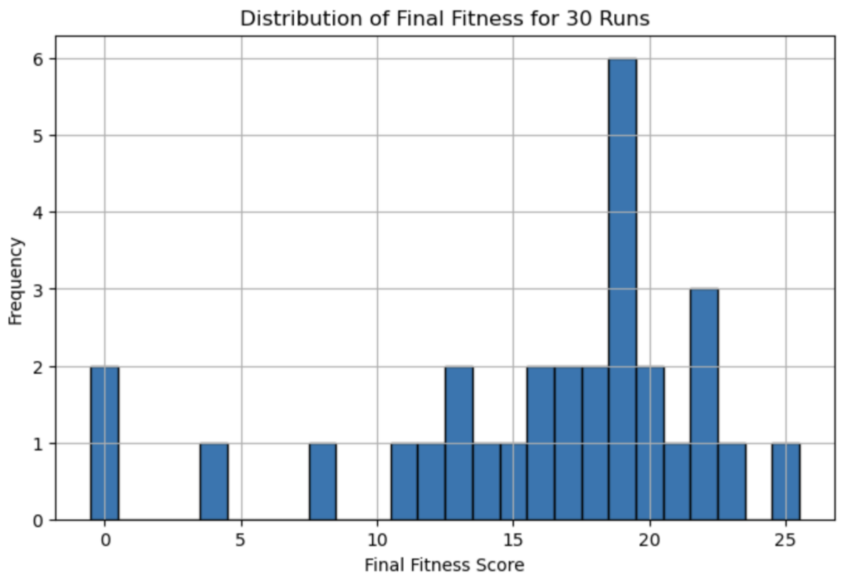

Figure 2: Distribution of final fitness scores across the 30 runs.

E Appendix: Assessment of Improvement Heuristics

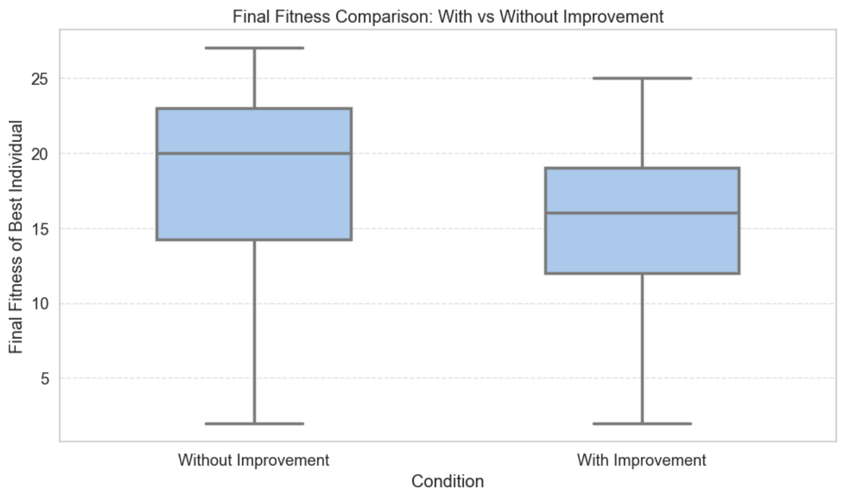

Figure 3: Boxplot of final fitness scores (with and without improvements).