Optimizing fitness center capacity with Genetic Algorithms

Matheus Felisberto [**NOVA IMS** | ***20230585@novaims.unl.pt***]

Keywords: Genetic Algorithms, Optimization, Charles, Mutation, Crossover, Selection
Group: 42
Github: https://github.com/mtfelisb/CIFO

1 Context

It is widely known among fitness enthusiasts that gyms often reach peak capacity at certain times, turning what should be an enjoyable workout into a stressful experience. Common solutions include training at different times or, more drastically, finding a new gym nearby. However, these alternatives are not always feasible due to time constraints like work or study hours, the distance to other fitness centers, and even the cost of a whole new subscription.

This project aims to identify the best solution using a given dataset of gyms and user preferences. By leveraging genetic algorithms, it suggests the optimal training schedule to enhance the overall gym experience.

2 Data

The dataset includes four fitness centers and their capacities at three different times of day: morning, afternoon, and evening, as described in Table 1. Additionally, it features 14 users, each with a preferred time slot (morning, afternoon, or evening) and a preferred gym (one of the four mentioned), as sampled in Table 2. The dataset also provides information on the distance in kilometers between each user and each gym as shown in Table 3.

Gym	Morning	Afternoon	Evening
Fitness24	3	2	3
Crossfit Gym	2	2	2
GymX	3	2	3
StrongFit	1	1	1

Table 1. Sample of gyms dataset

User	Preferred Time	Preferred Gym
João	Evening	Fitness24
Maria	Evening	Fitness24
Cecília	Evening	Crossfit Gym
Antonio	Evening	Crossfit Gym
Julieta	Evening	Fitness24

Table 2. Sample of users dataset

User	Fitness24	Crossfit Gym	Others...
João	1.2	3.0	...
Maria	1.0	2.8	...
Cecília	2.3	2.6	...
Antonio	3.2	1.0	...

Table 3. Sample of distances

3 Representation

In order to encode and decode this dataset for optimization using Genetic Algorithms, I followed the most common solution: binary encoding. However, a straightforward binary encoding would not suffice in this case, as encoding both a preferred gym and a preferred time in binary requires more bits. Specifically, I used two bits for the gym and two bits for the time preference. For example, if my preference was the last fitness center from Table 1, StrongFit, and my preferred time was evening, the encoding function would return '1110'. Here, the first two bits '11' represent the gym, and the last two bits '10' represent the time. '11' in decimal is 3, which corresponds to the index of StrongFit in the gym dataset, while '10' in decimal is 2, the index of evening in the dataset.

Its important to highlight that while effective, there is a fundamental problem with this encoding making unsuitable for bigger datasets. In fact, adding one more fitness center to the dataset would make stop working. The reason is that I'm using two bits to represent the gym, however, two bits can only represent up to four gyms, adding one more would encode an additional bit, making the decode function breaks. The solution to that would be implementing pairs of integers as representations. In the provided example, it would be (3, 2), being the indexes of StrongFit, and evening as preferred time respectively. This solution is still permutation based, hence everything would keep working smoothly, and even opening more mutation algorithms possibilities.

4 Fitness Function

As important as the representation, the fitness function requires extra care in its design. I based it on two pillars: penalties and rewards. In simple terms, penalties are applied when the capacity of a given fitness center is exceeded, and rewards are given when user preferences

are matched. Since I modeled it as a maximization problem, the higher the score, the better.

However, a problem with this design was it resulted in negative scores, which is problematic for selection algorithms such as Fitness Proportionate. To ensure all values are positive, I needed a method to shift all values upward by a constant. This constant was calculated based on the worst-case scenario, including the highest capacity penalty (users times penalty weight) and the largest distances times the distance weights. By shifting the values in this way, I eliminated negative scores while preserving the relative order of the scores. The complete calculation is given as the Equation 1.

$$\mathrm{F}(I) = C + \sum_{u \in U} \left(P(u,t) - w_c \cdot E(u,g,t) - w_d \cdot D(u,g)\right) \tag{1}$$

where:

- U represents all users.
- w_d is the distance penalty weight.
- w_c is the weight penalizing extra capacity usage.
- C is the shift constant calculated as $C = w_c \cdot U + w_d \cdot \max(\max(D))$.
- $P(u,t)$ scores 1 if the chosen time t for user u matches their preference, and 0 otherwise.
- $E(u,g,t)$ increments by 1 for each unit capacity exceeded at gym g during time t.
- $D(u,g)$ is the distance from user u to gym g.

5 Solutions

To implement a robust solution, given the dataset size and available algorithms, I decided to exhaustively run all possible combinations to find the optimal one and analyze it thoroughly. I highlighted the algorithms I implemented myself, while the others are already available within the Charles library.

Selection	Crossover	Mutation
Ranking	**Two Point**	Swap
Tournament	Single Point	Inversion
FPS	**Uniform**	**Scramble**

Table 4. Available algorithms

To systematically test 27 configurations (the permutations among the available algorithms), I created a function to run the genetic algorithm with the specified selection, mutation, and crossover algorithms. This function allows for parametrization of the number of runs, generations, and population size. With a small modification to Charles, the evolve method now returns the best fitness score per generation, which I use to calculate the median across the runs. The median is used instead of the mean to mitigate the impact of outliers, as suggested by Vanneschi and Silva [2023].

The Table 5 shows the configuration that each one of the 27 combinations ran on.

ID	Runs	Generations	Pop. Size	Elitism
1	20	20	20	True
2	20	100	20	True
3	20	100	40	True
4	100	100	40	True
5	100	100	40	False

Table 5. Configurations

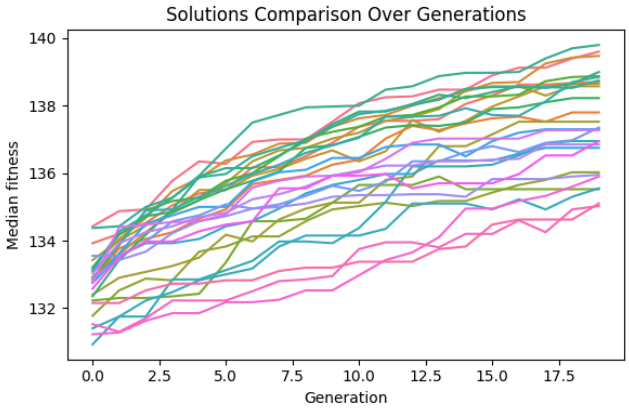

Figure 1. Configuration 1

6 Evaluation

I saved every run configuration presented in the Section 5 into a csv file, and then I used a notebook to evaluate every solution, to find the best algorithm combination, the worse, the effect of elitism, population size, generations and so on.

6.1 Configuration 1

The first solution, as described in Table 5 counts with 20 runs in total, 20 generations, 20 in population size and elitism. As a first run, I was just discovering values to be further fine tuned. I could conclude based on Figure 1 that almost every solution were still converging, in a tendency of going up. My first thought, then, was to increase the generation size.

6.2 Configuration 2

As the Subsection 6.1 only includes 20 generations, my first instinct was to increase this number to improve the solutions, aiming to better approximate a global optimum. Figure 2 shows that increasing the number of generations indeed helped the convergence of all solutions, as indicated by the curve. I ask the reader not to worry about identifying specific solutions in the plot yet, as I will drill down into this later. However, there are still solutions showing a tendency for growth, so I decided to increase the population size to see if it could further aid in convergence, thereby finding the best solution.

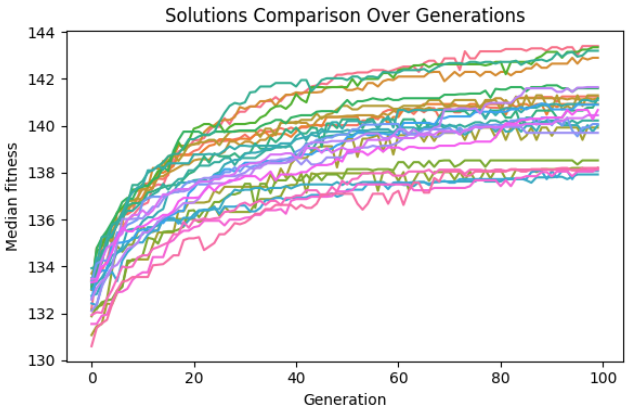

Figure 2. Configuration 2

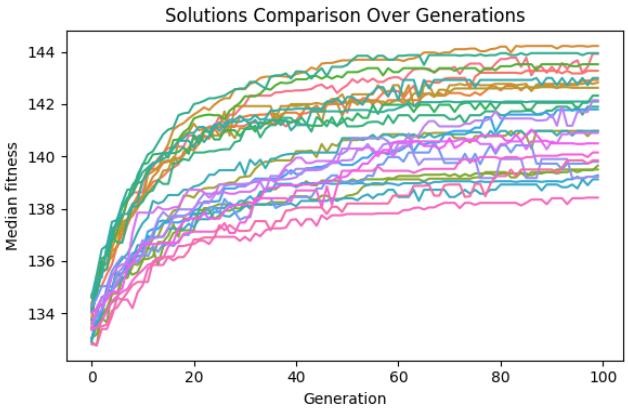

Figure 3. Configuration 3

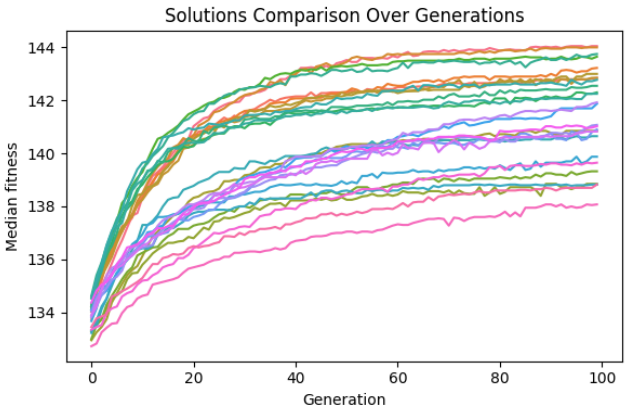

Figure 4. Configuration 4

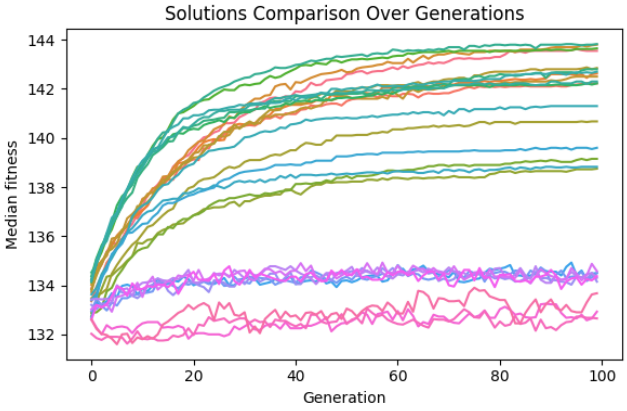

Figure 5. Configuration 5

6.3 Configuration 3

After doubling the population size, the computational time increased, but it is noticeable that although the top Y-axis values of configurations 2 and 3 are the same, the bottom values are not. This indicates that the solutions using a larger population size have better fitness even in the earlier generations, as shown in Figure 3. There is yet another noticeable difference between Figure 2 and Figure 3, but given the crispy aspect of the plot, it was difficult to define. Hence, I decided to increase the number of runs.

6.4 Configuration 4

Increasing the number of runs helped stabilize the lines in the plot, allowing for a more thorough analysis of the results. Based on Figure 4, I could identify two groups of solutions: one that converges faster and the other that takes more time. The first group contains the best solutions overall, characterized by the highest median fitness, while the latter group consists of the worst solutions.

6.5 Configuration 5

Additionally, I investigated the use of elitism within this problem. For the last configuration, I used the exact configuration from Subsection 6.4, but without elitism. The result, shown in Figure 5, demonstrates the impact of not using elitism across the 27 combinations and further highlights the two groups identified in Subsection 6.4. Essentially, without elitism, these solutions cannot evolve effectively through the generations. But the reader might be intrigued, why?

6.6 Worse Solutions

The group of worse solutions has only one thing in common: they use Fitness Proportionate as their selection algorithm. The problem illustrated by Figure 5 could be due to premature convergence, loss of high-fitness individuals, or a flat fitness landscape. Elitism helps improve convergence by preserving the best individuals and balancing exploration and exploitation.

On the other hand, the worse solutions when using elitism, regardless of selection or mutation method, performed poorly when using the Uniform Crossover method. However, they were still significantly better than FPS without elitism.

6.7 Best Solutions

The good stuff, finally. As I explained in Subsection 6.2, I would drill down into the solutions. The top 3 solutions and their combinations are represented in Table 6.

At first glance, it is observable that the best-performing mutation method for this problem was the Swap Mutation, used in all of the top 3 solutions. Fig-

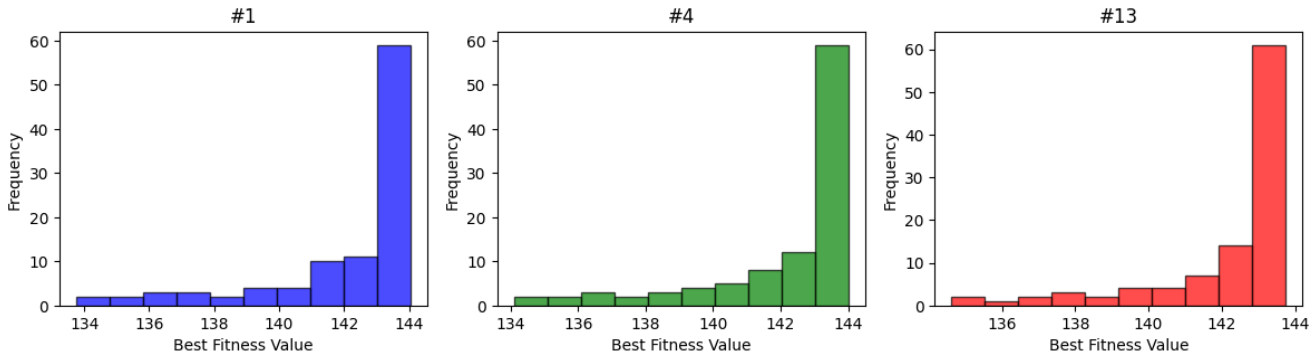

Figure 6. Best Solutions Data Distribution

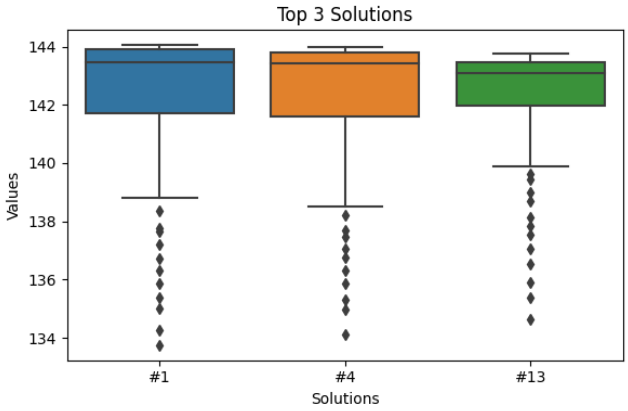

Figure 7. Best Solutions

ure 7 compares the performance of these 3 solutions. I understand the outliers at the bottom to represent the initial generations, where convergence was still occurring. The medians of solutions 1 and 4 are nearly the same, while the interquartile range difference with solution 13 is visible. To decide statistically which one is better, I first needed to determine the distribution of their data, whether it is normal or not.

The Figure 6 shows that rather than normally distributed, the data is left-skewed. Hence, the test I used was the Mann-Whitney U. The test reveals that, statistically, the top 3 solutions are actually a top 2. The p-values for solution id 1 vs solution id 13 and solution id 4 vs solution id 13, was 0.003 for both cases, and being smaller than 0.05, indicate strong evidence against the null hypothesis, suggesting that the distributions of these pairs are significantly different. Conversely, the p-value for solution id 1 vs solution id 4 is 0.57, providing no evidence to reject the null hypothesis and indicating that these two solutions are statistically the same.

ID	Selection	Crossover	Mutation
1	Ranking	Two-Point	Swap
4	Ranking	Single-Point	Swap
13	Tournament	Single-Point	Swap

Table 6. Best solutions combinations

7 Future Work

Many parameters remain to be further explored, such as mutation probabilities, crossover probabilities, and the weight penalties in the fitness function. The Tournament selection, which is part of the elite group of best solutions, also has a parameter that warrants further investigation. The representation issue presented in Section 3 is another important topic to explore. Additionally, there are hundreds, if not thousands, of permutation-based algorithms that could be implemented. A larger dataset would also be beneficial to continue improving the robustness of the solution. Additionally, I would further investigate the implementation of Pareto Optimization (multi-objective optimization) as it seems appropriate for this problem. However, changes would need to be made to the core of Charles to support it, as well as my fitness function implementation described in Section 4. Finally, assessing the computational time required for each combination would also be an interesting topic to explore.

8 Conclusion

Through this work, I have come to understand the crucial importance of good representation and a well-designed fitness function. While Genetic Algorithms can be fine-tuned through selection, mutation, and crossover, they cannot perform effectively without a good representation. If the fitness function is poorly designed, suboptimal solutions like those discussed in Subsection 6.6 will prevail. There is no definitive right or wrong solution, but rather an exploration of different algorithms. In cases where exhaustive testing of combinations is not feasible, certain decisions, such as avoiding the use of FPS without elitism, should be made judiciously, as it is evident that such configurations will lead to suboptimal solutions without needing to run the tests.

References

Buontempo, F. (2019). *Genetic Algorithms and Machine Learning for Programmers.* Pragmatic Bookshelf.

Jacobson, L. and Kanber, B. (2015). *Genetic Algorithms in Java Basics*. Apress.

Rocca, M. L. (2021). *Advanced Algorithms and Data Structures*. Manning Publications.

Vanneschi, L. and Silva, S. (2023). *Lectures on Intelligent Systems*. Springer.

Wirsansky, E. (2020). *Hands-On Genetic Algorithms with Python*. Packt Publishing.