Optimization Algorithms

Project Report

Optimizing route planning with GAs

Lourenço Passeiro - 20221838

Miguel Marques -20221839

Peter Lekszycki - 20221840

Tomás Gonçalves - 20221894

19th May 2024

NOVA Information Management School

Instituto Superior de Estatística e Gestão de Informação

Universidade Nova de Lisboa

Abstract

This work intends to investigate the optimization of the in-game currency earnings in the video game “Hollow Knight” using Genetic Algorithms. The objective was to determine the most efficient path through the game's ten areas while maximizing Geos.

The methodology involves generating and evaluating candidate solutions, represented as permutations of the game locations while respecting specific constraints. Multiple selection algorithms, genetic operators, and other parameters were tested on various datasets. Elitism was also considered to maintain high-quality solutions across generations. Our datasets, formatted as lists of lists, represent different Geo values and were created under controlled conditions to ensure validity and robustness. Parameter optimization was conducted using different types of searches, in a complementary way. Our final model, based on their performance across multiple datasets, demonstrated robust results, outperforming other combinations of parameters.

This approach highlights the efficacy of GAs in solving complex problems within dynamic environments like video games.

Introduction

Everybody already spent many hours playing video games and thinking about the fastest way to develop their character. It's in our nature as humans to want the best with the least amount of effort possible—a quicker path. Imagine, then, that you are immersed in the fascinating world of a well-known video game called Hollow Knight. In this game, your goal is to meticulously explore ten different areas, each of which presents chances for you to retrieve the game's currency, known as Geo. So, the issue that has to be answered is: out of all the several paths you may take in the game, which one yields the highest Geo earnings?

Addressing this question involves more than just trial and error. It requires a systematic approach that can efficiently explore the vast solution space and identify the optimal route. Genetic Algorithms are useful in this situation, by mimicking the process of natural selection to find solutions to complex problems. By iteratively generating and selecting candidate solutions based on their fitness, GAs can effectively search for optimal solutions in large and complex solution spaces.

We will then proceed with the implementation of several different algorithms, with different combinations of parameters, on several datasets, respecting the constraints imposed on us, that will end with choosing the most robust algorithm.

Methodology

Given that we have a permutation-based problem, we had to make sure that every step would respect the problem's characteristics while maintaining efficiency. We took into consideration the Knapsack code provided, and with research we modified it and systematically applied new concepts, to generate the best and most robust results possible.

Datasets

All our datasets are in the form of matrices, on which each index corresponds to an area of the game. The departure area was always the first index, and arrival the second one.

To create the datasets, we started by defining a function (*generate_matrix*), that receives as input boundaries for the Geo and returns a matrix with values inside them. When creating, we also made sure that the matrix-related constraints were respected:

- In each matrix, there will always be more Geo gains than losses. If not, the process of generating Geo earnings recursively went back to the beginning, and the range shifted 1 unit to the right, to increase the probability of having more Geo gains than losses (if this shift was not done, it would be possible to fall on an infinite loop).
- The value of going from G to FC is always going to be at least 3.2% lower than the smallest Geo gain.

```
D    FC    G    QS    QG    CS    KS    RG    DV    SN
D ['-', 76, 82, 65, 46, 65, -96, 18, 39, 62]
FC [-41, '-', 92, -76, 38, -80, -87, 54, -99, -96]
G [87, -94.89, '-', 96, 76, -36, 8, -10, 30, 4]
QS [-60, -1, 91, '-', -67, -81, -96, 95, -66, 32]
QG [-78, 34, -18, 14, '-', 78, 25, -54, 42, 30]
CS [-90, -12, 59, 34, 81, '-', -6, 17, -99, -82]
KS [81, -71, -31, 85, -34, 89, '-', -31, 70, 1]
RG [8, -45, 23, -88, 6, 55, -44, '-', 92, 31]
DV [8, 3, -51, -35, -22, -58, 76, 70, '-', -6]
SN [-92, -90, -80, 97, -3, 20, 100, 93, -54, '-']
```

Fig. 1 - Example of a valid dataset respecting the given constraints

Individuals

Our solutions are represented by lists on which each element is a code (index) representing an area of the game.

To create them, we defined the function *generate_solution*, to create individuals of the same size (in our case, it was always 11 - the 10 areas, with Dirtmouth repeating at the start and the end). It does that by shuffling a list with the possible moving areas (from 1 to 9) and merging it in the middle of a list containing [0, 0]. By doing that, we enforced that each solution was initialized already starting and ending at the index 0 (Dirtmouth).

Before an individual was generated, it would also be tested by another function (*check_constraints)*. This function was always applied when generating and evaluating solutions, with special importance when we apply the operators. The role of it is to check if the following constraints were met:

- All sessions begin and end at Dirtmouth *****;
- If we skipped a location, it could only be King's Station, and only after and if we had gone directly from Queen's Station to Distant Village ;
- City Storerooms cannot be visited right after Queen's Gardens;
- The Resting Grounds can only be reached in the last half of the session.

***** Even though the constraint "All sessions must begin and end in Dirtmouth (D)", was always enforced when initializing a solution, we thought that we had nothing to lose when checking it again on *check_constraints*.

If any of these constraints were violated, the entire process of generating a solution would have to be repeated, until a submitted candidate solution would be considered valid.

After generating our solutions, we had to evaluate them. Our fitness function *get_fitness* simply receives as input an individual and a matrix and computes the sums of the Geos earned from moving between areas. We made sure that if we had skipped a location it wouldn't affect the Geos, or yield an error.

Again, if the inputted individual was not valid (checked by *check_constraints*), they automatically would be assigned a fitness of -1, which would practically disqualify them from being selected for the algorithm. Even though this will never happen in the first generation of the population (since we already check on *generate_solution*) this will be essential to penalize if the genetic operators create invalid solutions, which happens very often.

Population

At this stage, we simply applied the function created to generate and evaluate the individuals, to a population.

To evaluate the fitness of a population, we defined a function that, given a matrix and a population, computes the fitness of each individual. Here, we also checked that, in the case of DV being visited right after QS, we had an advantage by skipping KS. If we had, we would do so.

Selection Algorithms

We implemented the three selection algorithms presented in class: fitness proportional selection (roulette wheel), ranking selection, and tournament selection. These will be tested in our parameter search (with different values for tournament size, in tournament selection).

Genetic Operators

This section introduces Variation, a pillar of Natural Selection. By applying genetic operators to individuals within the parent population, we are enabling GAs to better explore the search space.

We implemented four different crossovers and five mutations: using multiple operators allows us to introduce different types of genetic diversity into the population, which can help avoid premature convergence to local optima and explore a wider range of solutions.

In each operator, we forced that the first and last index would be always 0 (simply iterated from the first to the penultimate index), to decrease the chance that the offspring would be invalid. However, if they would violate any of the problem's constraints, as said before, they would be assigned a very low fitness, practically disqualifying them from the evolution.

It is important to ensure that each operator brings a certain level of variance without compromising the complete structure of the individual. In other words, the mutation should not be so chaotic as to lose important segments, but it should be sufficient to avoid premature convergence to a local optimum.

Crossovers

Here, we implemented three well-known crossovers and created by ourselves one.

Before applying each crossover, we checked if any of the parents had skipped a location. If they had, we would replace the skip mark with the skipped location (KS). We did so because it is impossible to do our crossovers with elements in one parent without correspondence in the other parent. Nonetheless, in our algorithm, after the crossovers, we would still check if the individuals allowed to skip, had advantages to do so.

- Cyclic Crossover (CX)
  - The algorithm starts by randomly choosing a starting index, on which we will copy the gene from the parents to their respective offsprings.

Parents	Offsprings
0 2 4 8 3 6 7 1 9 5 0	0 - - - - - 7 - - - 0
0 1 9 7 6 5 4 3 8 2 0	0 - - - - - 4 - - - 0

  - The next chosen index will be the position of one of the genes of the last iteration, in the other parent.

Parents	Offsprings
0 2 4 8 3 6 7 1 9 5 0	0 - 4 - - - 7 - - - 0
0 1 9 7 6 5 4 3 8 2 0	0 - 9 - - - 4 - - - 0

  - We will repeat this until the chosen index is equal to the first position visited, closing the cycle.

Parents	Offsprings
0 2 4 8 3 6 7 1 9 5 0	0 - 4 8 - - 7 - 9 - 0
0 1 9 7 6 5 4 3 8 2 0	0 - 9 7 - - 4 - 8 - 0

  - In the end, we have to fill the offsprings with the genes of the unlinked parent, ensuring each offspring inherits genetic material from both parents.

Parents	Offsprings
0 2 4 8 3 6 7 1 9 5 0	0 1 4 8 6 5 7 3 9 2 0
0 1 9 7 6 5 4 3 8 2 0	0 2 9 7 3 6 4 1 8 5 0

- Partially Mapped Crossover (PMX)
  - The operator starts by randomly choosing a subset, that will be copied from the parents to their unlinked offsprings.

Parents	Offsprings
0 2 4 8 3 6 7 1 9 5 0	0 - - 7 6 5 4 - - - 0

0 1 9 7 6 5 4 3 8 2 0 | 0 - - 8 3 6 7 - - - 0

- To fill the offsprings, we try to copy the parent's genes to their respective offsprings. However, if the current gene is already on the offspring, we search for it and choose the gene of the other parent in that position. We do this process until we find a gene that is not on the offspring.

Parents | Offsprings

0 2 4 8 3 6 7 1 9 5 0 | 0 2 8 7 6 5 4 - - - 0
0 1 9 7 6 5 4 3 8 2 0 | 0 1 9 8 3 6 7 - - - 0

- We repeat this process until the offspring are filled.

Parents | Offsprings

0 2 4 8 3 6 7 1 9 5 0 | 0 2 8 7 6 5 4 1 9 3 0
0 1 9 7 6 5 4 3 8 2 0 | 0 1 9 8 3 6 7 5 4 2 0

- <u>Ordered Crossover (OX1)</u>
  - It starts by copying a subset of the parents to their unlinked offsprings.

Parents | Offsprings

0 2 4 8 3 6 7 1 9 5 0 | 0 - - 7 6 5 4 - - - 0
0 1 9 7 6 5 4 3 8 2 0 | 0 - - 8 3 6 7 - - - 0

  - To fill the offsprings, we get the genes from their respective parents, that are not in the offsprings, in their natural order:

On parent 1, not on offspring 1: 2,8,3,1,9

On parent 2, not on offspring 2: 1,9,5,4,2

  - Then we fill them on the offspring, by the order they were on the parents, starting from the last crossover point. When we reach the end of the individual we just go back to the beginning.

Parents | Offsprings

0 2 4 8 3 6 7 1 9 5 0 | 0 1 9 7 6 5 4 2 8 3 0
0 1 9 7 6 5 4 3 8 2 0 | 0 4 2 8 3 6 7 1 9 5 0

- Two-Point Crossover
  - This crossover was our implementation of the two-point crossover, normally used in problems with binary individuals, to our permutation-based problem.
  - It starts by copying a subset of the parents to their unlinked offsprings. The rest will remain unchanged.

Parents	Offsprings
0 2 4 8 3 6 7 1 9 5 0	0 2 4 7 6 5 4 1 9 5 0
0 1 9 7 6 5 4 3 8 2 0	0 1 9 8 3 6 7 3 8 2 0

  - Now, the offspring may have duplicated areas. To fix that, we first select the duplicates of each offspring:

Duplicates on offspring 1: 5, 4

Duplicates on offspring 2: 8, 3

  - Then, we iterate over the duplicated values in the offspring subset and swap them, to have unique genes.

Parents	Offsprings
0 2 4 8 3 6 7 1 9 5 0	0 2 4 7 6 8 3 1 9 5 0
0 1 9 7 6 5 4 3 8 2 0	0 1 9 5 4 6 7 3 8 2 0

Mutators

- Swap Mutation
  - This operator iterates all over the positions of the individual and when activated chooses 2 random positions.

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 2 4 8 3 6 7 1 9 5 0

  - Then, we swap the selected positions:

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 2 4 9 3 6 7 1 8 5 0

- We repeat this process until we reach the final swappable position of the individual.

- Inverse Mutation
  - This operator iterates all over the positions of the individual and when activated chooses 2 random positions. The genes between them will constitute a segment.

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 2 4 8 3 6 7 1 9 5 0

  - Then, we invert that segment:

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 2 4 9 1 7 6 3 8 5 0

  - We repeat this process until we reach the final selectable position of the individual.

- Mix Mutation
  - This operator iterates all over the positions of the individual and when activated chooses 2 random positions. The genes between them will constitute a segment.

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 2 4 8 3 6 7 1 9 5 0

  - Then, the order of the genes on that segment is randomly shuffled.

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 2 4 1 9 3 8 6 7 5 0

  - We repeat this process until we reach the final selectable position of the individual.

- Displacement Mutation

- This operator iterates all over the positions of the individual and when activated chooses 2 random positions. The genes between them will constitute a segment.

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 2 4 8 3 6 7 1 9 5 0

- Then, we insert that segment in a different position, displacing it.

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 2 4 1 9 5 8 3 6 7 0

- We repeat this process until we reach the final selectable position of the individual.

- <u>Two-Point Mutation</u>
  - This mutation can be considered more disruptive, since it works in groups of two positions, and swapping them, possibly leading to very different solutions. It has its benefits but also could lead to loss of information.
  - This operator iterates all over the positions of the individual.

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 2 4 8 3 6 7 1 9 5 0

  - Then, we choose two random positions, and they will form, with the gene ahead of them, two duos. These two groups cannot overlap themselves.

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 2 4 8 3 6 7 1 9 5 0

  - After this, we swap them:

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 9 5 8 3 6 7 1 2 4 0

  - It continues this process until we reach the last valid positions.

Individual	Mutation
0 2 4 8 3 6 7 1 9 5 0	0 9 5 6 7 8 3 1 2 4 0

Elitism

The presence of elitism in GAs enables the best solutions to ensure that their influence stays for subsequent generations. It can help converge towards optimal or near-optimal solutions in complex landscapes. However, this can sometimes lead to a lack of diversity in the population, as the best individuals dominate successive generations, limiting the exploration of the search space.

To introduce the possibility of elitism into our problem, we defined the function *get_n_elites* that given a population and their fitnesses selects the top *n* fittest individuals. This pre-defined *n* can be any number from 1 to *pop_size*.

Algorithm

Our algorithm joins all the elements created before. We create a population (*initializer*) of *pop_size* length, evaluate it (*evaluator*), and start a cycle that is going to be repeated *n_gens* times.

In each generation, we create a new population of offsprings, that are created by selecting (by the *selector*) some members of the past population (the parents), and applying our genetic operators (*crossover*, *mutation*) to them. Each genetic operation has a probability of being performed (*p_m*, *p_xo*), and if it is not performed, the individuals pass to the next generation without changes (reproduction). We stop adding and creating new offsprings when their size reaches the *pop_size*. If we are using elitism, we are going to keep the *n* best individuals of the current population into the next one.

At the end of each generation, the offspring population becomes the new population, and we check if there is a benefit by skipping KS, giving some constraints. Then, we retrieve the best individual found.

Results

Here, we tested multiple parameter combinations in order to get the best model possible. To ensure that we tested our models well enough, we defined 18 different geo matrices that will be input to every model.

These have six different magnitudes - from two to seven digits. For each magnitude, we created three different types of matrices:

- Balanced positives - their initialization sets the likelihood of our Geos values being positive at 51%, subject to probabilistic outcomes;
- Majorly positives - their initialization sets the likelihood of our Geos values being positive at 70%, subject to probabilistic outcomes;
- Overwhelming positives - their initialization sets the likelihood of our Geos values being positive at 90%, subject to probabilistic outcomes.

Initially, we truly tried to exhaustively try every parameter combination, but later we realized that it was too computationally expensive for our computers. Given this, we had to seek other options, that would still be good enough to seek the best combination of parameters possible. The found and chosen method was Bayesian Optimization, a sequential model-based optimization technique that balances exploration and exploitation to efficiently find the best model, without having to test all possibilities.

Firstly, we did some searches to find the best range of parameters. Our final parameter ranges:

- Selecting Algorithms (5): Roulette Wheel, Ranking Selection, Tournament Selection with tournament size equal to 2, 5, and 10;
- Crossover Operators (4): Cyclic Crossover, Partially Mapped Crossover, Ordered Crossover and Two-Point Crossover;
- Mutator Operators (5): Swap Mutation, Inversion Mutation, Mix Mutation, Displacement Mutation and Two-Point Mutation;
- Population Size (3): 25, 50, 100
- Number of Generations (3): 250, 500, 750
- Crossover Probability (4): 0.2, 0.4, 0.6, 0.8
- Mutation Probability (4): 0.2, 0.4, 0.6, 0.8
- Presence of Elitism (2): True, False
- Number of Elites (if Elitism) (4): 2, 3, 4, 5

With our possible parameters defined, we can start to search for the optimal combination. Our strategy was to get some candidate models with Bayesian optimization, and then test them in our 18 different geo matrices. The model with the better scores across all of them would be selected.

We performed the Bayesian optimization 10 times for each matrix, thus getting 10 candidate models for each matrix. In total, we got 5 different models.

To evaluate our results, we created the following measures: 'Votes', representing the number of matrices that each combination was suggested by during the search; 'Wins', representing the number of times that each model had the highest score in some matrix; 'NetVotes', representing the votes of each model that did not win.

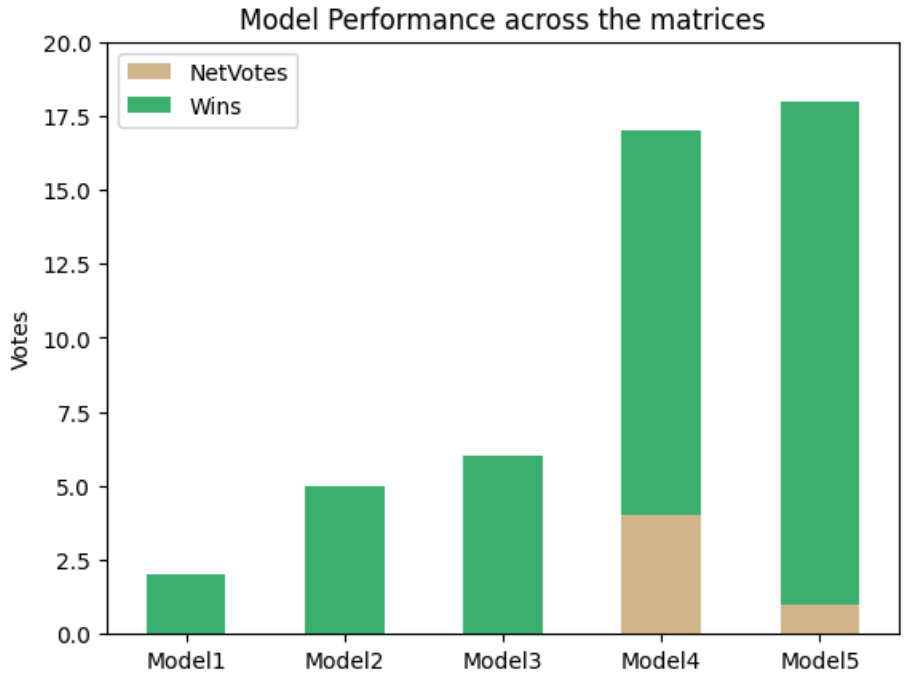

Fig.2 - The different models yielded by the Bayesian optimization, evaluated by our performance measures

By this, we can see that two models seem to be the best. Nonetheless, we will test all of the candidate models, to test their robustness across the matrices. Each model was run 5 times for each matrix, and their score is the mean of these values.

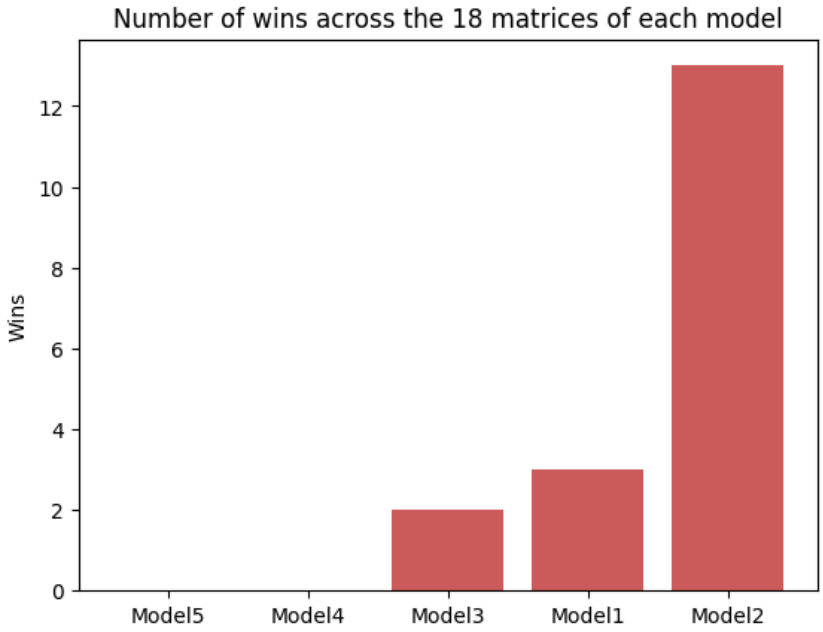

Fig.3 - The different models compared by number of wins

As we can see, the models with worse performance in the Bayesian Optimization, surprisingly dominated the others across every matrix. To reach a conclusion about our best model, we also ranked our matrix scores (as a form of scaling), and computed the mean and median score of each model.

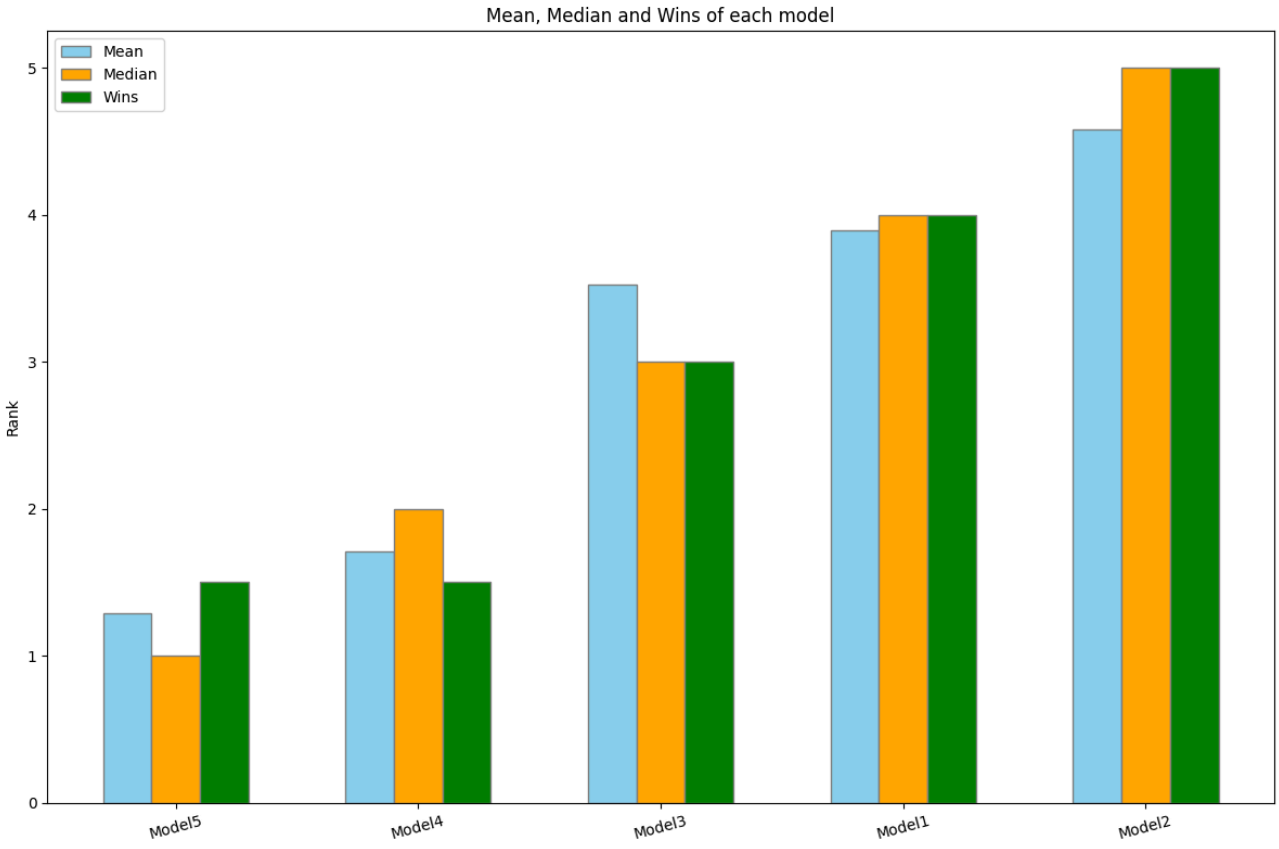

Fig.4 - The models compared by our performance measures

As expected, the model with the most wins also dominated every other metric. Given this results, the Model 2 will be our final choice of a model. Its parameters:

- Selector: Tournament Selection with a tournament size of 10
- Crossover Operator: Two-Point-Crossover
- Mutation Operator: Swap Mutation
- Population Size: 25
- Number of Generations: 500
- Crossover Probability: 0.8
- Mutation Probability: 0.2
- Elitism: False

Conclusion

This study employed a comprehensive methodology, including the generation of valid datasets, the application of various selection algorithms, crossover, and mutation operators, and parameter tuning to find the final optimal model. The final optimized model proved to be the most robust across diverse datasets.

We had to overcome several issues, such as the well-implemented generation of heterogeneous datasets, respecting every constraint given to us. The process of creating crossovers and mutations was also laborious. As it was a permutation-based problem, we had to adapt the concepts given to us in class, also knowing that this was a crucial stage in the work. Throughout the work, we ran the experiments several times to reduce the level of randomness in the study. Since the initial data is randomly generated, it is essential to perform multiple runs to ensure that the results obtained are consistent and representative.

Nonetheless, we managed to find an ideal route. The candidate solutions emerged using Bayesian Optimization, with also testing them on every one of our matrices. It was possible at this stage of the work due to a smaller number of combinations and guaranteed us greater confidence in the results.

This study has replicability in various study areas and real-world scenarios. The absence of specific and known initial data reflects a common situation in many real problems, where we often do not have all the necessary information from the beginning. In such situations, it is essential to find local rather than global optimal solutions, which are often impossible to determine due to the complexity and size of the search space.

To conclude, it was quite interesting to carry out this study because it presented several complex challenges and required us to always be attentive to details. This experience not only expanded our understanding of genetic algorithms and their practical applications but also highlighted the importance of systematic and adaptive approaches in solving complex problems.

References/Annexes

[1] Leonardo Vanneschi. Optimization Algorithms course materials.

“Lectures on Intelligent Systems”, Vanneschi, Silva

https://www.sciencedirect.com/topics/computer-science/crossover-operator

https://www.rubicite.com/Tutorials/GeneticAlgorithms/CrossoverOperators/CycleCrossoverOperator.aspx

https://mat.uab.cat/~alseda/MasterOpt/GeneticOperations.pdf

https://dergipark.org.tr/tr/download/article-file/534290

https://xloptimizer.com/features/genetic-algorithms-ga/genetic-algorithm-for-permutation-problems-permga

https://scikit-optimize.github.io/stable/modules/generated/skopt.gp_minimize.html

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação

Universidade Nova de Lisboa