OPTIMIZATION ALGORITHMS

FINAL PROJECT REPORT

Duarte Carvalho 20221900
Laura Matias 20221836
Luca Passera 20231932
Matilde Casimiro 20221940

Contents

Abstract

In this project, a genetic algorithm (GA) was employed to address an optimization challenge within the video game *Hollow Knight*. The primary objective is to enhance the player's gaming experience by maximizing the accumulation of the in-game currency, Geo, through an optimized route.

The project's **background** lies in a thorough exploration of various genetic operators used in the project, including selectors, crossovers, and mutators, which are designed to optimize the adventurer's path in the game world by evaluating different potential routes.

A structured approach is adopted for the **methodology**, which includes formulating the problem, implementation of the genetic algorithm, and the execution of a *Grid Search* to identify the optimal parameters.

The **results** section is dedicated to presenting and analyzing the effectiveness and robustness of the proposed solution. The ultimate achievement of the project, a single, optimal route that the player will consistently follow in the game, will be highlighted.

In the **conclusion**, reflections on the insights gained and challenges encountered during the project are shared. The significance of Genetic Algorithms in tackling complex optimization tasks is underscored, and potential avenues for future research are discussed.

Overall, this project demonstrates the practical application of Genetic Algorithm optimization in gaming scenarios, testing both implementation skills and creativity.

Introduction

In society nowadays, the need for effective solutions to solve problems is more important than ever. Enhancing the gameplay experience in *Hollow Knight*, particularly in optimizing the path to maximize Geo (the in-game currency) earnings, becomes essential. This involves selecting routes between game areas that yield varying amounts of Geo.

To address this challenge, a genetic algorithm was employed - an optimization technique inspired by the principles of natural evolution. Genetic algorithms simulate the process of natural selection and genetics through the use of mutations and crossover operators, which are applied to a population of individuals to find the optimal solution, each representing a potential solution to the problem at hand, with the aim of identifying the optimal solution.

This report explores how genetic algorithms can identify the best route to maximize Geo earnings in *Hollow Knight*, considering that different game areas offer varying amounts of Geo and present risks that can result in Geo loss.

To accomplish this, numerous functions were developed across different files to execute various tasks within the genetic algorithm. These tasks include generating potential solutions, applying different genetic operators, and more. This comprehensive approach ensures a thorough and effective solution to the optimization challenge at hand.

Background

Selectors

Selection algorithms are vital in evolutionary processes as they determine which individuals from a population move on to the next generation. These algorithms simulate the process of natural selection, in which individuals with desirable characteristics have a better chance of surviving and passing on their genes to the next generation.

In this project, five selection methods were considered: ***Roulette Wheel Selection***, ***Tournament Selection***, ***Self-Adaptative Tournament Selection***, ***Linear Ranking Selection***, and ***Exponential Ranking Selection***.

Roulette Wheel Selection

Roulette Wheel Selection is a selection operator where each individual's probability of being selected is directly proportional to its fitness relative to the rest of the population. The selection probability for each individual is calculated as follows:

$$P(\mathrm{sel}\ i) = p_i = \frac{f_i}{\sum f_j}$$

The probability is determined by dividing the fitness of the individual by the sum of the fitness values of all individuals in the population.

As a result, individuals with higher fitness levels have a greater chance of being selected. This process resembles a roulette wheel, where each individual occupies a segment of the wheel proportional to its selection probability. During the selection process, the wheel is spun, and two parents are selected based on these probabilities.

Tournament Selection

Tournament Selection operates by selecting individuals from a population based on their fitness values using a tournament-style approach. This method involves choosing a random subset of individuals (determined by the tournament size) from the population. Among these participants, the one with the highest fitness is selected as the tournament winner. This process is repeated to select multiple individuals, ensuring that those with higher fitness have a greater chance of being chosen. Ultimately, the two parents are selected from the tournament winners.

Self-Adaptive Tournament Selection

The **Self-Adaptive Tournament Selection** process begins by calculating the diversity of the population, which involves computing the average fitness, determining the fitness range, and quantifying the deviation of individual fitness values from the average. The diversity value, which ranges between 0 and 1, influences the tournament size dynamically. The formula used for adaptive tournament size is:

$$\text{Adaptative Tournament Size} = 2 + (\text{N} - 2) \times \text{diversity}$$

Here, *2* is the minimum tournament size, *N* is the population size, and the *diversity* factor adjusts the tournament size based on the population's diversity. When *diversity* is low, the tournament size is close to 2, and when *diversity* is high, the size approaches *N*. This dynamically adjusted tournament size is used to select a pair of individuals from the population, balancing selection pressure based on diversity.

Once the tournament size is determined, the selection process proceeds identically to traditional Tournament Selection. A random subset of individuals is chosen, and the one with the highest fitness within this subset is selected as the tournament winner. This process is repeated to select multiple individuals, with the two parents ultimately selected from the tournament winners.

Linear and Exponential Ranking Selection

Both **Linear** and **Exponential Ranking Selection** methods begin by sorting the population in descending order based on fitness values to maximize fitness. Each member of the sorted population is then assigned a rank, starting from 1 for the fittest individual and increasing accordingly.

After ranking, the selection probability for each individual is computed based on its rank. In **Linear Ranking Selection**, a selection pressure parameter (between 1 and 2) is used to calculate the probability. The probability of selecting an individual is determined by:

$$P(\text{sel } i) = p_i = \tfrac{2-s}{N} + 2 \times \tfrac{(i-1)(s-1)}{N(N-1)}$$

where *s* is the selection pressure parameter and, *N* is the population size.

In **Exponential Ranking Selection**, an exponential function is used to compute the selection probability, with a constant parameter controlling the shape of the probability distribution. The probability of selecting an individual is given by:

$$P(\text{sel } i) = p_i = (1 - e^{-\frac{\text{ranks}}{k}})$$

where *ranks* is the rank of the individual, and *k* is the constant controlling the distribution's shape.

These methods ensure that individuals with higher ranks (i.e., better fitness) have a higher chance of selection while incorporating an element of randomness. The rate of increase in selection probability is controlled by the respective parameters in each method.

Finally, both methods directly select two parents from the ranked population based on the calculated probabilities.

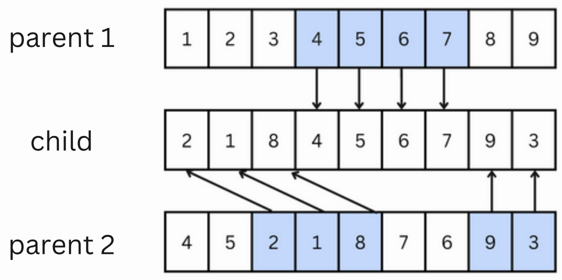
Crossovers

Crossover operators are essential as they generate new candidate solutions, by exchanging genetic material at specific points, crossover promotes diversity in candidate solutions, aiding in the search for optimal solutions.

In this project, five crossover methods were considered: ***Order Crossover, Position-Based Crossover***, ***Cycle Crossover***, ***Partially-Mapped Crossover***, and ***Modified Partially-Mapped Crossover***.

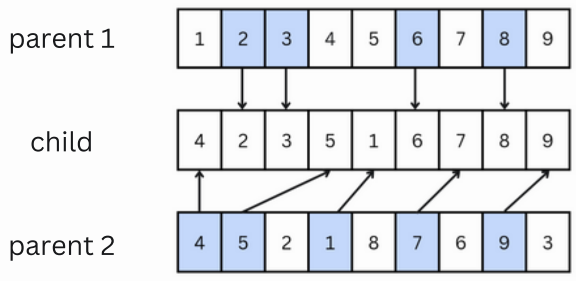
Order Crossover

Order Crossover involves selecting two random crossover points within the parent routes. The parents are then divided into segments based on these crossover points. During the crossover, a segment between the crossover points is inherited from one parent, preserving the order and position of the elements. The remaining elements are taken from the other parent, maintaining their original order. This process ensures that no duplication of elements occurs. Finally, the process is repeated to create two offspring.



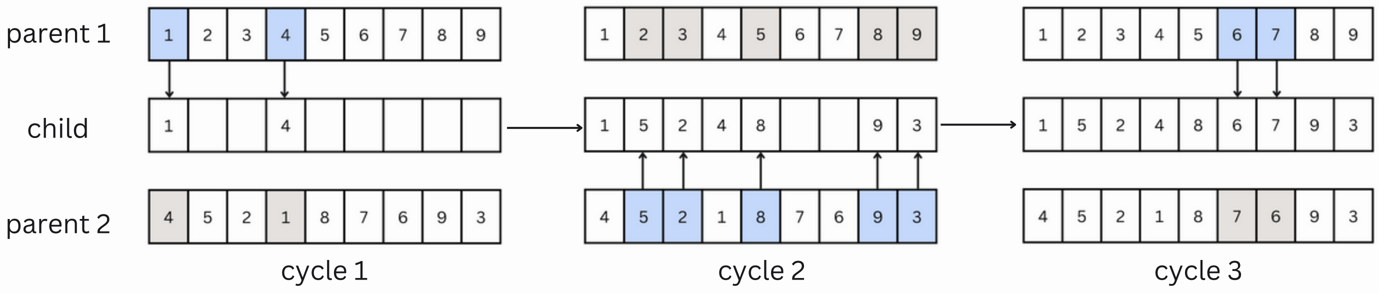
Position-Based Crossover

Position-Based Crossover selects specific crossover points within parent routes. The offspring inherit elements from one parent at these designated points, maintaining the same position as in the parent. Then, the remaining positions are filled with elements from the second parent that are not already present in the offspring, also maintaining the same order but in different positions. Finally, the process is repeated to create two offspring.



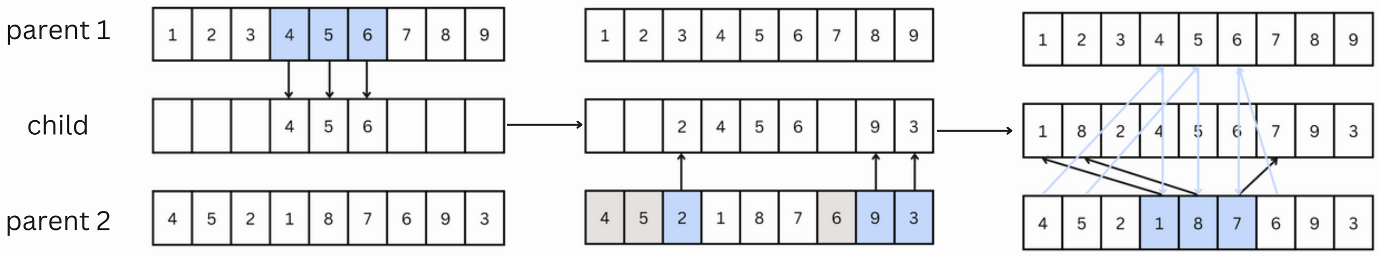
Cycle Crossover

Cycle Crossover begins by selecting a random starting index in the parent chromosomes, which, in this case, is zero. The gene from the first parent at this position is copied to the child, and the corresponding gene from the second parent is located. This process of copying genes from one parent and locating their corresponding genes in the other parent continues until the cycle is completed (i.e. when the starting index is encountered again). The remaining positions in the child are filled with genes from the second parent that are not already present. Finally, the process is repeated to create two offspring.



Partially-Mapped Crossover (PMX)

In **Partially-Mapped Crossover** (PMX), two crossover points are randomly selected. The segment between these points is copied directly from the first parent to the child. The remaining positions are filled using a mapping between the two parents: for each gene from the second parent that is not yet present in the child, its corresponding gene from the first parent is found, and the gene from the second parent in that position is placed in the child. This process ensures that no gene is duplicated in the child, maintaining the integrity of the genetic material. Finally, the process is repeated to create two offspring.



Modified Partially-Mapped Crossover

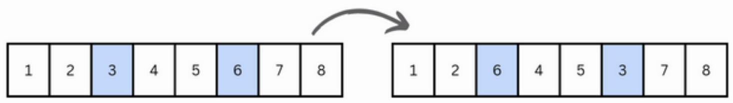
Modified Partially-Mapped Crossover is similar to PMX, but after copying the segment between the crossover points from the first parent and the elements of the second parent in the corresponding position, the remaining positions are randomly filled with genes from the first parent that are not yet present in the child. This approach may introduce more diversity in the offspring compared to PMX, as it randomizes the filling of remaining positions. Finally, the process is repeated to create two offspring.

Mutations

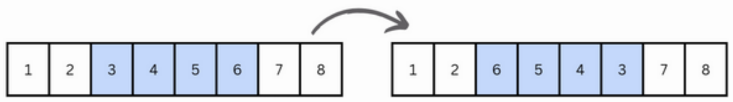
Mutations are fundamental for exploring the solution space, introducing random modifications to candidate solutions. These modifications help avoid local optima, allowing the algorithm to explore an even broader spectrum of potential solutions.

In this project, five mutator methods were considered: ***Swap Mutation, Inversion Mutation, Scramble Mutation, Displacement Mutation*** and ***Thrors Mutation.*** For each, a copy of the individual to be mutated is made, and if a randomly generated number is below the mutation rate, the mutation occurs.

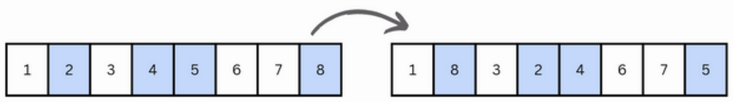
- **Swap Mutation**: Two distinct positions within the individual are randomly chosen, and the elements at these positions are swapped.



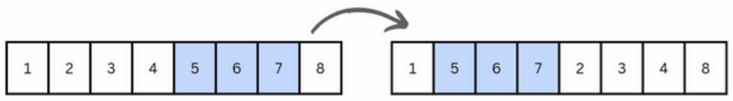
- **Inversion Mutation**: Two distinct positions are randomly chosen, defining a segment within these positions. The elements within this segment are then inverted, meaning their order is reversed.



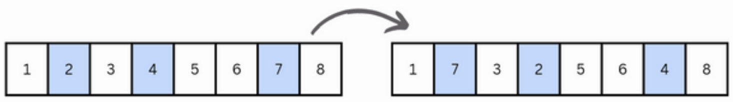
- **Scramble Mutation**: Random positions within the individual are chosen, and the elements at these positions are shuffled randomly.



- **Displacement Mutation**: A random segment within the individual is removed and reinserted at a random position within the individual.



- **Thrors Mutation**: Three random positions within the individual are selected, and the elements at these positions are cyclically rotated: the element at the first position moves to the second position, the element at the second position moves to the third position, and the element at the third position moves to the first position.



Methodology

As previously mentioned, a genetic algorithm was implemented to achieve the best possible route in the game *Hollow Knight*. Genetic algorithms are optimization techniques that replicate natural selection by iteratively selecting, crossing, and mutating candidate solutions, based on a fitness function, to find optimal or near-optimal solutions to complex problems. The implementation used is summarized in the following flowchart:

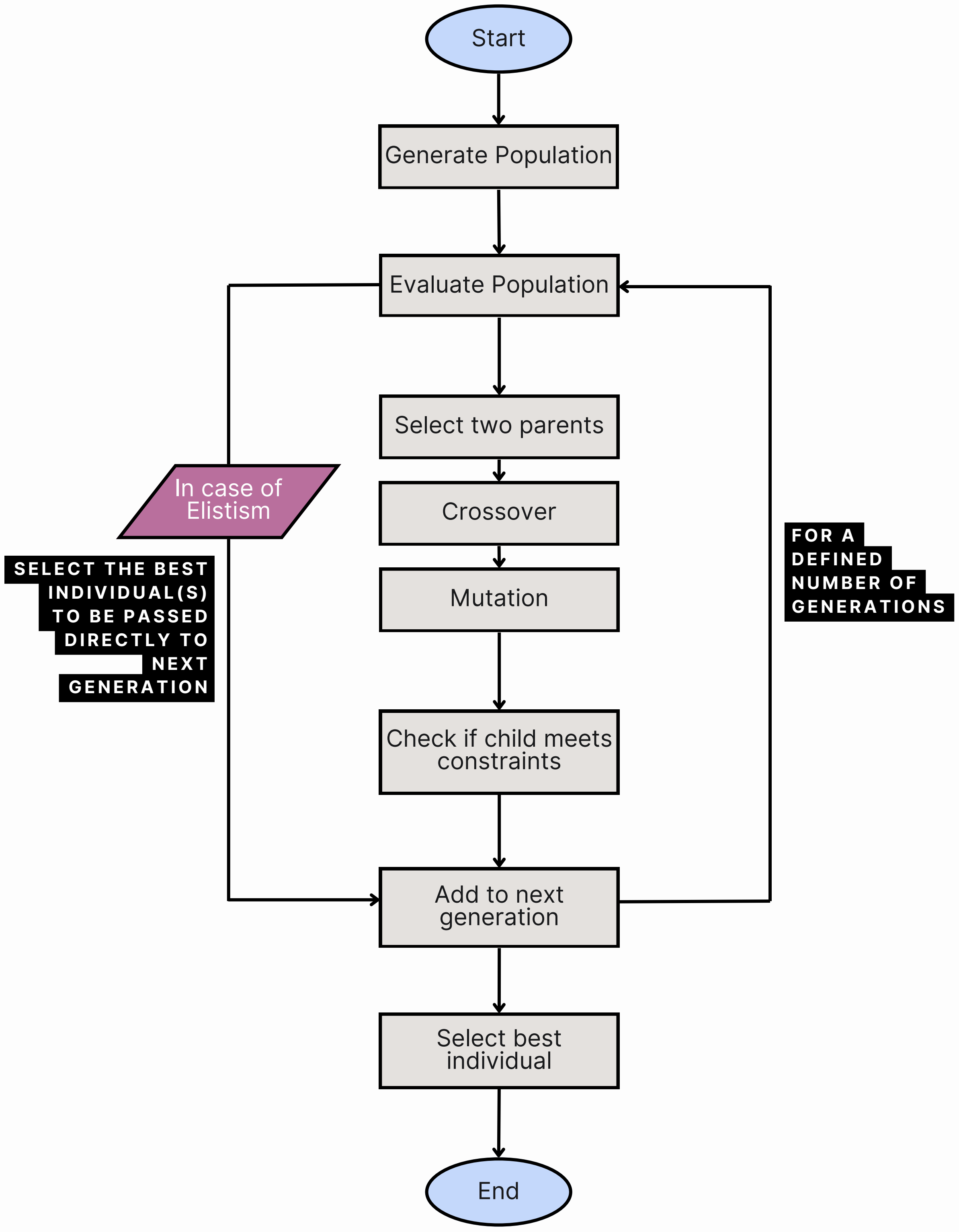


fig. 1 - Flowchart of the genetic algorithm process

Initializers

Generation of Individuals

In the context of the algorithm for the video game *Hollow Knight*, the generation of individuals, which represent routes, is a crucial step. This process involves defining an appropriate solution structure that adheres to the specific rules of the game. To achieve this, two main functions were created: ***generate_individual*** and ***check_contraints.***

The ***generate_individual*** function is responsible for creating random representations of routes within the game. These routes are simplified to a list of strings, where each string represents an abbreviation of one of the ten areas in the game. For instance, *Dirthmount* is abbreviated as 'D', *Forgotten Crossroads* as 'FC', and so forth. Thus, an individual consists of a sequence of 11 areas, with ‘D’ being the designated start and end of every route. This mirrors the mandatory constraint that every game session should start and end at *Dirthmount*.

However, there are additional constraints that the individuals should follow:

- Each area can only be visited once in a route, excluding the first and last positions.
- ‘CS’ (*City Storerooms*) cannot directly succeed ‘QG’ (*Queen’s Garden*).
- ‘RG’ (*Resting Grounds*) must be visited during the second half of the route.
- Each route must go through all the areas in the game, with the exception of the following constraint.
- In the case that ‘DV’ (*Distant Village*) is placed directly after ‘QS’ (*Queen’s Station*), ‘KS’ (*King’s Station*) may be excluded from the route.

In light of these constraints, the ***check_constraints*** function was developed. This function’s purpose is to verify whether any constraint is violated within a given route. If a violation is detected, the function returns **True**, indicating that the route is not valid. Conversely, if all constraints are satisfied, the function returns **False**, meaning that the route is valid. Additionally, this function is tasked with the eventual removal of the area ‘KS’ from a route, if ‘DV’ follow ‘QS’ and such alteration results in a higher scoring route. In these cases, ‘KS’ is replaced by ‘PH’ (Placeholder), since the crossover operators implemented within the algorithm need parents of equal size to function correctly.

Within the ***generate_individual*** function, a shuffling mechanism is implemented to create the individuals, excluding ‘D’. After this, the ***check_constraints*** function is systematically called while the individuals produced are not acceptable. Once the constraints are verified, ‘D’ is added as the first and last position of the route. This process ensures that all generated routes obey the game’s specified rules.

Generation of Population

Once individual routes are generated, a population is assembled using the ***generate_population*** function, with a specified population size (pop_size) as argument. This function iteratively calls ***generate_individual*** to create a population of valid individuals, repeating the process pop_size times.

Fitness Evaluation

After the individuals were created, the ***route_geo_gains*** function was developed to calculate the total fitness, or Geo gains, for a given route. It takes two arguments: individual, which represents the individual route, and points_matrix, a matrix (list of lists) representing the points gained by moving from each area to all the other areas. A valid example of a points matrix can be found in the Annexes.

The function iterates through the individual route, calculating the gains obtained by moving from one area to the next based on the provided points matrix. If 'PH' is present in the individual, it is excluded from the calculation, as it doesn't represent any area, and is, as the name states, a placeholder to make sure all individuals in the population have the same length. The total geo gains obtained from that route are then returned. The gains and, therefore the fitness values, are integer values.

Subsequently, the fitness of each individual in the population is assessed using the ***evaluate_population*** function. This function takes the population of individuals and a points matrix as parameters. It utilizes the previously defined ***route_geo_gains*** function to calculate the fitness of each individual, and compiling these fitness values into a list.

Algorithm

The developed function to execute the genetic algorithm begins by generating an initial population and calculating its corresponding fitness values. Then, a selection function is employed to choose two distinct parents (the selection process is repeated for a maximum of 5 iterations, if the parents are equal) to reproduce via crossover operations (with a specified probability rate), producing two offsprings. Following this, a mutation operator function (also subject to a specified probability rate) was applied to the offsprings. In both crossover and mutation operations, the first and last positions of the routes, represented by 'D', were always excluded from these operations, as every route needs to start and end in *Dirthmount*.

At this point in the reproduction process, the ***check_constraints*** function is called to verify whether the offsprings obey all the game rules. Since the size of the offspring population needs to be equal to the size of the initial population, this approach is repeated until a set of offsprings that doesn't violate any constraints is generated.

The process repeats until it reaches the specified number of generations. Optionally, it provides information regarding each generation as it is running (verbosity), logs the results to a *csv* file and plots its fitness landscape. Finally, it returns the best individual found and its fitness value. As this is a maximization problem, the best individual is the one with the highest fitness value, or in this case, the route with the highest Geo gains.

The ***genetic_algorithm*** function takes as input the following parameters:

- **initializer**: The function to create the initial population.
- **pop_size**: The size of the population.
- **points_matrix**: The matrix representing the points gained by moving from each area to all the other areas.
- **fitness_evaluator**: The function to evaluate the fitness of the population.
- **generations:** The number of generations to evolve the population.
- **crossover_operator**: The crossover operator for generating offspring individuals.
- **mutator**: The mutation function for offsprings.
- **selector**: The selection function to select parents to reproduce.
- **ts_size**: Tournament size, in case tournament selection is the selector.
- **elite_size**: Quantity of best individuals to preserve in the next generation.
- **verbosity**: Whether to provide information and outputs during the optimization process.
- **p_xo**: The probability of performing crossover.
- **p_m**: The probability of performing mutation.

- **plot**: Whether to plot fitness landscape.
- **seed**: Initial value used by random number generator.
- **log**: Whether to log the results to a csv file.

Elitism

Elitism in genetic algorithms is a mechanism used to ensure that the best individuals from each generation are preserved and carried over to the next generation without alteration. This helps maintain high-quality solutions within the population and prevents the loss of optimal or near-optimal solutions due to random variations introduced by the genetic operators. This way, elitism usually accelerates convergence towards an optimal solution and improves the overall performance of the genetic algorithm. However, there can be some disadvantages associated with elitism, such as premature convergence, where the algorithm quickly settles on initially good solutions, limiting further exploration of the solution space, which might prevent the algorithm from discovering new, potentially better solutions.

Consequently, balancing elitism with mechanisms that maintain diversity, such as the crossover and mutation operators is crucial for the effective performance of a genetic algorithm. Additionally, the choice of the correct elite size, which refers to the number of top-performing individuals that are preserved in the next generation, is also important to ensure that the best solution is achieved by the algorithm.

Probability of Genetic Operators

Setting a random seed ensures that the sequence of random numbers used to determine whether crossover and mutation occur remains the same across different runs of the algorithm.

The probabilities of crossover and mutation occurring are specified by parameters p_x and p_m, respectively. However, the actual decision of whether to perform crossover or mutation at a particular step is made based on randomly generated numbers. For example, to decide whether to perform crossover on a pair of parent individuals, the algorithm generates a random number between 0 and 1, and if this number is less than or equal to the crossover probability (p_xo), crossover is performed; otherwise, it is not.

By setting a random seed, the sequence of random numbers generated by the random number generator is fixed. This way, we can guarantee that, although the probabilities of crossover and mutation are defined and constant, the outcomes of these decisions are consistent across different runs, leading to reproducible results.

Grid Search

In order to maximize the potential of the genetic algorithm, it is highly important to discover the best combination of parameters to use as input in the function, thereby improving its performance and effectiveness in solving the optimization problem at hand.

Grid Search is a popular and traditional method for hyperparameter optimization. It works by making an exhaustive search based on a defined set of parameters, where every combination of parameters is computed, and the their performance is calculated based on a chosen metric.

To this end, the function ***grid_search*** was developed, along with two helper functions, ***evaluate_combination*** and ***clean_combinations***.

The ***evaluate_combination*** function is responsible for assessing the performance of a given combination of parameters. It takes in a dictionary of parameters (params), a callable function (algorithm), the number of iterations to run the set of parameters (iterations), and a list of data matrices for each iteration (points_matrices). Inside this function, a try-except block ensures that any potential errors during the evaluation are caught and handled gracefully, so that the process can carry on. The function calls the algorithm with the specified parameters and records the results for the number of specified iterations, updating the params dictionary with the current points_matrix from points_matrices for each iteration. It then calculates the average fitness of the results and logs it to a CSV file. If an error occurs along the execution of the function, it prints an error message and returns a performance score of 0.

The ***clean_combinations*** function filters out redundant parameter combinations to optimize the grid search. It receives a list of all possible parameter combinations (combinations) and it iterates through each combination, applying specific criteria to determine redundancy. As the genetic algorithm function takes as input the parameter ts_size, even if the selector is not tournament_selection, when we are contructing the parameters to test, there will be combinations trying different tournament sizes for selection methods that don’t use this parameter. Having this in mind, if a parameter combination involves a selector that is not tournament_selection and has a ts_size that doesn’t equal 5, it is considered redundant and discarded. This step helps in reducing the computational load by avoiding unnecessary evaluations.

The ***grid_search*** function is responsible for the overall grid search process. It accepts a callable function (algorithm), the number of iterations to run each parameter combination (iterations), and a list of parameters to test (parameters).

The function begins by initializing a multiprocessing pool, a tool from the multiprocessing library, to parallelize the evaluation process, which speeds up the computation. By leveraging multiple CPU cores, the library allows the grid search to run multiple evaluations simultaneously instead of sequentially. Without multiprocessing, the grid search would evaluate each combination one after the other, which would be much slower, especially since we were dealing with a large number of combinations and iterations.

Following this, it generates all possible the combinations of parameters to test using the ***ParameterGrid*** function from the sklearn.model_selection library and filters them using the ***clean_combinations*** function. At this point, an informative message indicating the number of combinations to be tested is printed.

Next, the function generates the data matrices' list that will be used in the evaluation of each parameter combination by calling the ***generate_points_matrix*** function in a list comprehension.

A partially applied function (***partial_evaluator***) is created using the ***partial*** function from the functools module, which simplifies passing the algorithm, iterations, and points matrices parameters to the ***evaluate_combination*** function during the parallel processing, as the pool.map, the method used to evaluate all parameter combinations and collect the results, can only take one input parameter for the function it is parallelizing. Once all evaluations are complete, the results are sorted based on the average performance, with the highest-performing combination at the top, since this is a maximization problem. The best parameter combination and its corresponding performance are extracted and printed. Additionally, to manage memory efficiently, the function deletes unnecessary variables and triggers garbage collection. Finally, the multiprocessing pool is closed to release resources, and the best parameter combination is returned.

Test Data

In the context of the grid search, the generated matrices that serve as input data for the algorithm being explored, ensure that each parameter combination is evaluated using consistent and comparable data across iterations, leading to more accurate results in the hyperparameter optimization process.

Having this in the mind, the ***generate_points_matrix*** function creates a 10 by 10 matrix of random integer values between -500 and 500, representing the gains obtained by moving from each area to all the other. Is it also adjusted to make sure that the gain of moving from 'G' to 'FC' is at least 3.2% less than the minimum positive value in the matrix, since this condition is always present in the game.

The following parameters, which were organized into a dictionary, **ga_parameters**, were explored in the grid search:

PARAMETERS	OPTIONS TO TEST
initializer	generate_population
pop_size	20, 50, 100
points_matrix	gains_matrix
fitness_evaluator	evaluate_population
generations	5, 10, 20
crossover_operator	order_crossover, position_crossover, cycle_crossover, partially_mapped_crossover, modified_partially_mapped_crossover
mutator	swap_mutation, scramble_mutation, displacement_mutation, thrors_mutation, inversion_mutation
selector	roulette_wheel_selection, tournament_selection, self_adaptative_tournament_selection, linear_ranking_selection, exponential_ranking_selection
ts_size	5, 10, 15
elite_size	0, 1, 2
p_xo	0.8, 0.9, 0.95
p_m	0.05, 0.1, 0.2
verbosity	False
plot	False
seed	0, 1
log	False

tab. 1 - Table with parameters to test in grid search

This resulted in 85 005 combinations of parameters, that were tested 15 times each on the same set of data matrices to produce an accurate and reliable average fitness.

Results

After running the previously mentioned grid search, the results showed that the best combination of parameters is the following:

- Population Size (**pop_size**): 100.
- Generations (**generations**): 20
- Crossover Operator (**crossover_operator**): Position Crossover
- Mutator (**mutator**): Displacement Mutation
- Selector (**selector**): Tournament Selection
- Tournament Size (**ts_size**): 5
- Elite Size (**elite_size**): 0
- Crossover Probability (**p_xo**): 0.95
- Mutation Probability (**p_m**): 0.2
- Seed (**seed**): 1

This configuration ensures a thorough exploration of the solution space. The Position Crossover operator combines the positions of elements from the parents, while the Displacement Mutation moves a subsequence of genes to a new position within the individual. Tournament Selection with a tournament size of 5 was chosen to balance selection pressure and diversity. No elitism (elite size of 0) was used, which further enhances genetic diversity. The high crossover probability of 0.95 and the mutation probability of 0.2 promote significant recombination, while maintaining diversity. Using a seed of 1 guarantees the reproducibility of the results.

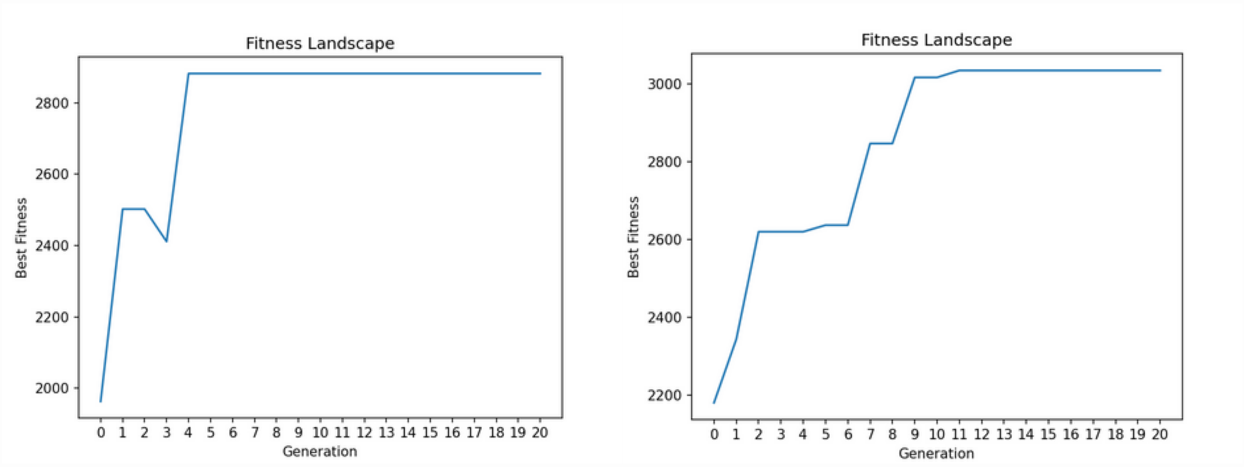


fig. 2 - Fitness Landscapes

The plots above illustrate the fitness landscape of two different runs of the algorithm, each using different random matrices generated by the ***generate_points_matrix*** function. These matrices incorporate the geo gains, highlighting the variations in fitness across different runs. Additional plots demonstrating further variations of the algorithm workflow can be found in the Annexes.

Conclusion

This project explored the use of genetic algorithms (GAs) for optimizing game routes in Hollow Knight. Through research and testing of various genetic operators, we observed the adaptability and effectiveness of GAs in complex optimization tasks. GAs demonstrated their ability to explore numerous parameter combinations, making them adaptable to diverse challenges. Their balance between exploration and exploitation was key in optimizing routes to maximize Geo earnings within game constraints.

The project emphasized the flexibility, scalability, and robustness of GAs as optimization tools. Their parallel execution, stochastic nature, and transparent optimization process makes them applicable to real-world scenarios beyond gaming.

The insights from this project confirm the effectiveness of GAs in optimizing gaming routes in *Hollow Knight*. Their utility extends beyond this problem, covering a broad range of scenarios where complex optimization is required.

In summary, this project not only deepened our understanding of GAs but also highlighted their potential in addressing diverse optimization problems. Through ongoing research and innovation, Genetic Algorithms continue to enhance decision-making and problem-solving in optimization.

References

[1] Vanneschi, L., & Silva, S. (2023). Lectures on Intelligent Systems. Retrieved from https://doi.org/10.1007/978-3-031-17922-8

[2] Karazmoodeh, A. (2023). Selections in Genetic Algorithms. Retrieved from Karazmoodeh, A. (2023). Selections in Genetic Algorithms. Retrieved from https://www.linkedin.com/pulse/selections-genetic-algorithms-ali-karazmoodeh-g9yyf/

[3] Karazmoodeh, A. (2023). Mutations in Genetic Algorithms. Retrieved from https://www.linkedin.com/pulse/mutations-genetic-algorithms-ali-karazmoodeh-u94pf/

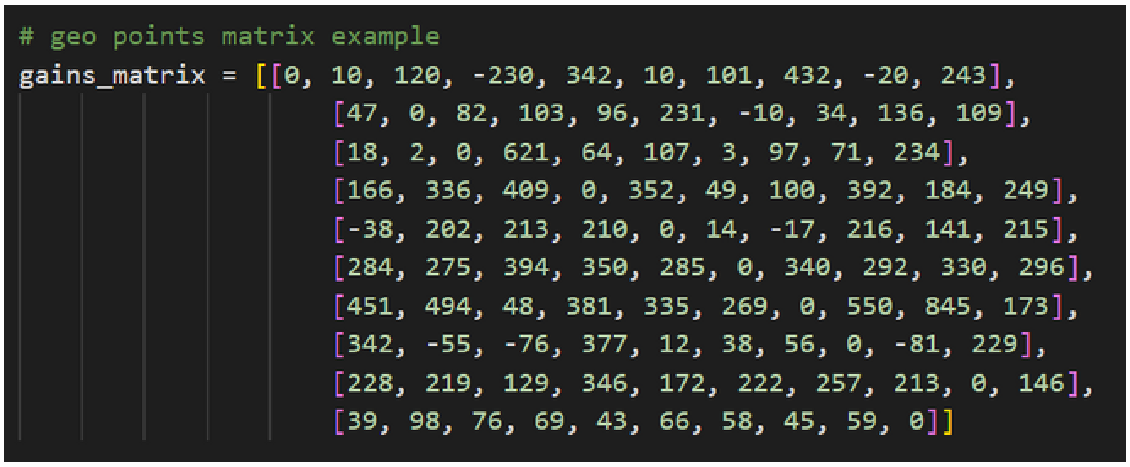
[4] Soni, K., & Kumar, M. (2013). Study of Various Mutation Operators in Genetic Algorithms. Retrieved from https://www.semanticscholar.org/paper/Study-of-Various-Mutation-Operators-in-Genetic-Soni-Kumar/31a4010f427f6c485c577504e08e895a34e054d2

[5] Pandey, H. M., Chaudhary, A., & Mehrotra, D. (2014). A Comparative Review of Approaches to Prevent Premature Convergence in GA. International Journal of Computer Science and Information Technologies, 5(3), 4166-4170. Retrieved from https://www.ijcsit.com/docs/Volume%205/vol5issue03/ijcsit20140503404.pdf

[6] Udayakumar, E., & Baskaran, R. (2015). Performance Analysis of Various Crossover Operators in Genetic Algorithms. ICTACT Journal on Soft Computing, 6(1), 1083-1092. Retrieved from https://ictactjournals.in/paper/IJSC_V6_I1_paper_4_pp_1083_1092.pdf

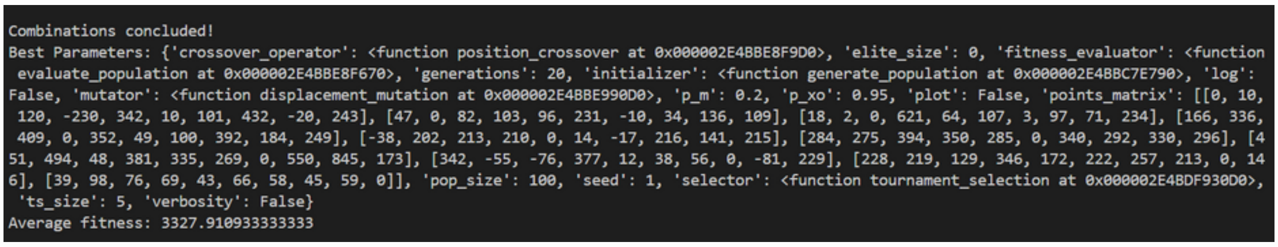
[7] Scikit-learn developers. (2024). sklearn.model_selection.ParameterGrid. In Scikit-learn 1.4.2 documentation. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.ParameterGrid.html

Annexes



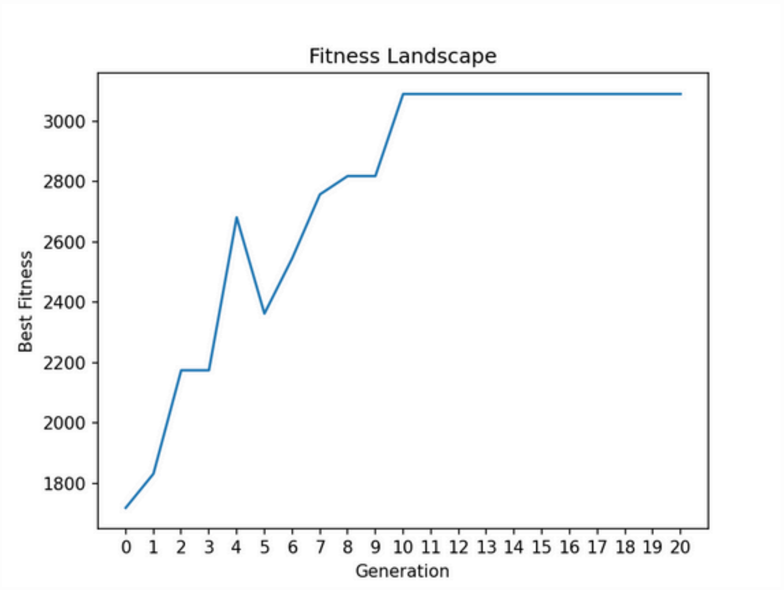
```python
# geo points matrix example
gains_matrix = [[0, 10, 120, -230, 342, 10, 101, 432, -20, 243],
                [47, 0, 82, 103, 96, 231, -10, 34, 136, 109],
                [18, 2, 0, 621, 64, 107, 3, 97, 71, 234],
                [166, 336, 409, 0, 352, 49, 100, 392, 184, 249],
                [-38, 202, 213, 210, 0, 14, -17, 216, 141, 215],
                [284, 275, 394, 350, 285, 0, 340, 292, 330, 296],
                [451, 494, 48, 381, 335, 269, 0, 550, 845, 173],
                [342, -55, -76, 377, 12, 38, 56, 0, -81, 229],
                [228, 219, 129, 346, 172, 222, 257, 213, 0, 146],
                [39, 98, 76, 69, 43, 66, 58, 45, 59, 0]]
```

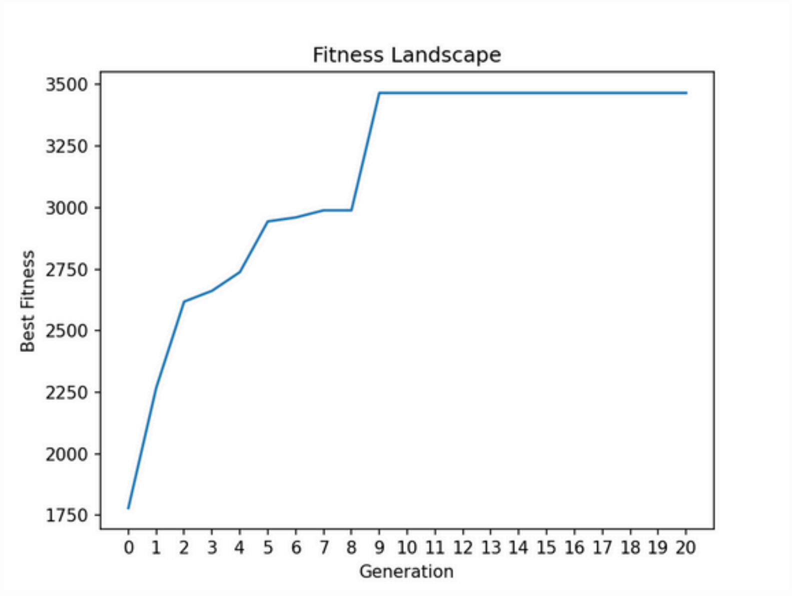
annex 1 - Geo Gains matrix example

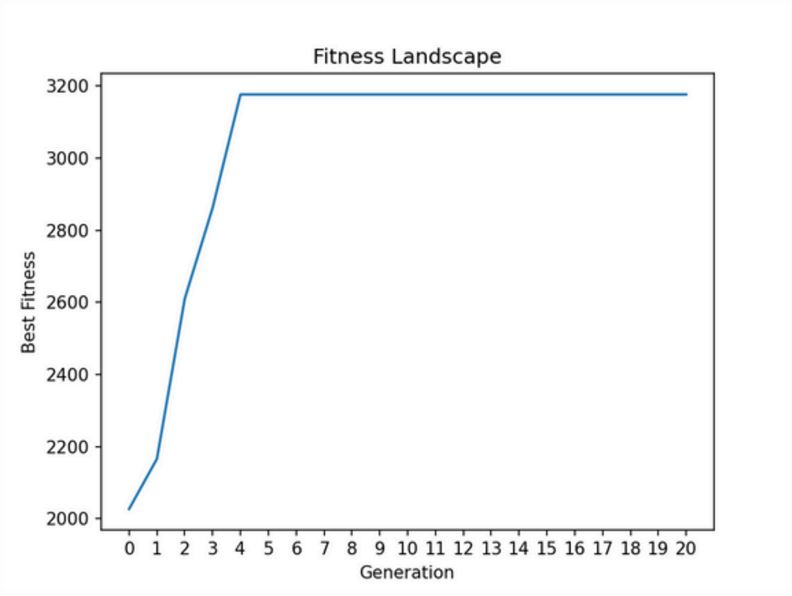


```
Combinations concluded!
Best Parameters: {'crossover_operator': <function position_crossover at 0x000002E4BBE8F9D0>, 'elite_size': 0, 'fitness_evaluator': <function
 evaluate_population at 0x000002E4BBE8F670>, 'generations': 20, 'initializer': <function generate_population at 0x000002E4BBC7E790>, 'log':
False, 'mutator': <function displacement_mutation at 0x000002E4BBE990D0>, 'p_m': 0.2, 'p_xo': 0.95, 'plot': False, 'points_matrix': [[0, 10,
 120, -230, 342, 10, 101, 432, -20, 243], [47, 0, 82, 103, 96, 231, -10, 34, 136, 109], [18, 2, 0, 621, 64, 107, 3, 97, 71, 234], [166, 336,
 409, 0, 352, 49, 100, 392, 184, 249], [-38, 202, 213, 210, 0, 14, -17, 216, 141, 215], [284, 275, 394, 350, 285, 0, 340, 292, 330, 296], [4
51, 494, 48, 381, 335, 269, 0, 550, 845, 173], [342, -55, -76, 377, 12, 38, 56, 0, -81, 229], [228, 219, 129, 346, 172, 222, 257, 213, 0, 14
6], [39, 98, 76, 69, 43, 66, 58, 45, 59, 0]], 'pop_size': 100, 'seed': 1, 'selector': <function tournament_selection at 0x000002E4BDF930D0>,
 'ts_size': 5, 'verbosity': False}
Average fitness: 3327.910933333333
```

annex 2 - Grid Search results, as they are displayed in Visual Studio Code







annex 3 - Fitness landscapes produced by the algorithm with the final parameters