OPTIMIZATION ALGORITHMS

PROJECT REPORT

Genetic Algorithms for problem solving

Francisco Gomes - 20221810
Maria Henriques - 20221952
Marta Almendra - 20221878
Vidhi Rajanikante - 20221982

2023/2024

Table of contents

Introduction

The pursuit of efficiency and in-game resource maximization is a recurring problem in the field of video game optimization. In this project, we explored the intriguing field of route optimization with regard to the well-known video game called "Hollow Knight."
Our mission was to create an algorithmic solution that optimized our friend's journey, assuring maximum Geo gains while reducing any losses.

There were certain special restrictions and goals specified for this task. Starting and finishing at Dirtmouth's area, our optimization strategy involves navigating 10 different game zones. Every region offers different chances for gaining Geo gains as well as possible dangers that might result in losing them.

To overcome this challenge, we employed a genetic algorithm, which consists of a computational method based on natural selection and evolutionary biology. In order to find optimal routes, we generated individual solutions that form a population and subjected them to selection, crossover, and mutation processes. In addition, a method to determine the optimal genetic algorithm parameters, making the algorithm more effective in solving the problem, was also employed and will be discussed later on.

Throughout this report, we will go over the approach and steps used to construct our algorithm, emphasizing critical factors, restrictions, and design decisions. Furthermore, we will offer empirical results and analysis which will demonstrate our algorithm's efficacy and performance.

Ultimately we hope that our project will be a useful example on how computational intelligence and gaming innovation may work together to advance the rapidly, yet developing field of video game optimization.

Background

We wanted to complete this project as efficiently as possible, so in addition to the strategies and techniques covered in the Optimization Algorithms course, we implemented a few methods that we learned through other sources. In this section, we have included a description of the different types of selection, crossovers and mutation processes utilized.

SELECTORS [1][2][3][4][5]

Selection algorithms function similarly to nature's selection, sifting through a population to select the very best of the crop for future generations. Just as in nature, where only the fittest survive to pass on their genes, these algorithms favor individuals with favorable traits, giving them a better chance of reproduction.

By focusing on individuals with higher fitness values, selection algorithms steer the genetic process towards more promising areas of the solution space. This helps refine and enhance solutions over time, much like how natural selection refines species over generations.

These algorithms work by assigning probabilities to individuals based on their fitness. Everyone has a chance to be picked, but those who are stronger contenders stand a better chance. And it's important to note that all parents chosen for reproduction come from the same population, ensuring genetic consistency as evolution unfolds.

ROULETTE SELECTION

Roulette selection is a stochastic selection strategy in which the likelihood of selecting an individual is proportional to their fitness. The system is inspired by real-world roulettes, although it has significant differences from them.

Let N be the number of individuals in population P (population size) and let F = [f1,f2,...,fN] be the set of their fitness values.

For any individual i ∈ P, the probability of selecting i, for maximization problems, is:

$$P(\text{sel } i) = p_i = \frac{f_i}{\sum_{j=0}^{N} f_j}$$

RANKING SELECTION

In Ranking Selection the individuals are chosen based on their relative fitness. All individuals are sorted according to their fitness scores, typically from worst to best; ' N ' is the rank given to the best individual, whereas rank '1' is assigned to the worst.

Contrary to Roulette Selection, each individual is assigned a given probability based on its rank, ensuring the selection pressure in more evenly distributed across the population. This prevents the fittest individuals from dominating the selection process and thus, maintaining the population diversity.

TOURNAMENT SELECTION

This method operates by breaking down the selection process into two stages. Initially, a number 'k' of individuals is chosen randomly from the population, ensuring this first part does not depend on fitness values and is purely random.

Once the individuals are selected, the process shifts to a deterministic phase where the individual with the highest fitness among the chosen 'k' is selected.

The parameter 'k', known as tournament size, plays a crucial role in controlling the intensity of the selection pressure. The selection pressure is the degree to which the fittest individuals are favoured: the higher the selection pressure, the more the better individuals are favored.

EXPONENTIAL RANK SELECTION

Similar to Ranking Selection, this method selects the individuals based on their rank rather than their raw fitness values.

Initially, the individuals are sorted in descending order based on fitness and each inidividual is assigned a rank; 1 for the highest fitness 'N' for the worst. Each rank is then used to calculate a probability using an exponential decay function. For each individual, the probability of being selected is given by the following function, where 'rate' is the parameter that controls selection pressure and 'i' the rank of the individual, is:

$$P(i) = e^{-\mathrm{rate} \cdot i}$$

The probabilities are then normalized to allow a valid probabilistic selection.

LINEAR RANK SELECTION

Linear rank selection is a technique in genetic algorithms that selects individuals from a population depending on their fitness levels. Instead of immediately utilizing fitness values to establish selection probabilities, individuals are first sorted based on their fitness values. Every individual is then given a rank depending on their position in the sorted list. Each individual gets a different rank even if their probabilities are same.

The selection probability for each individual is determined by their rank, with better-ranked individuals having a larger chance of being chosen.

This technique guarantees that individuals with lesser fitness have a chance of being picked while still preferring fitter ones, hence maintaining population variety and preventing premature convergence.

CROSSOVERS [4][6]

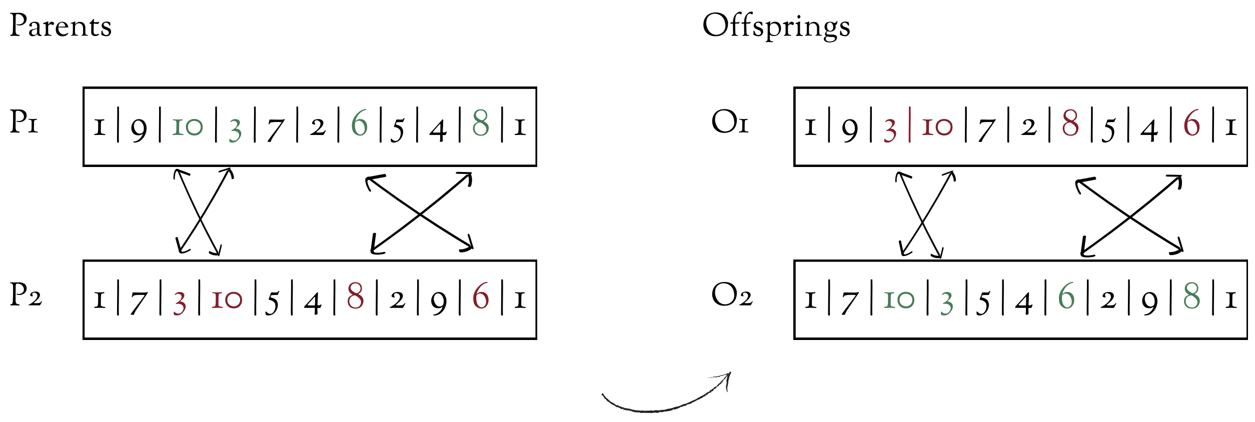
A crossover operator is used to produce offsprings by combining genetic material from two parents. These operators imitate the genetic recombination process that occurs during the reproduction process in living organisms. Crossovers involve selecting two parent individuals from the population and randomly selecting a crossover point or locations along their genetic sequences. The genetic material beyond the crossover points is exchanged between the parents, resulting in one or more children. The offspring acquire genetic qualities from both parents, integrating characteristics from each parent with the intention of developing better solutions.

CYCLE CROSSOVER

The cycle crossover is an operator often used for permutation problems such as ours, which is finding the most optimal route. By preserving the positions in the parent individuals, verifying that each value from the parents appears exactly once in the offspring is the principle behind the cycle crossover.

Iterating through the positions of the parents individuals, the values in the offspring individuals are filled with the corresponding values of the positions from the parent individuals, based on the identified cycles and this occurs through every cycle until all the positions in the offspring individuals are filled with the corresponding values from the parent individuals.

Below is an example of an individual before and after the cycle crossover.



UNIFORM CROSSOVER

The uniform crossover is an operator which is quite straightforward and aids in maintaining diversity amongst the individuals in our population.

It works by combining genetic information from two parent individuals to produce a new offspring individual. By having a binary mask of the same length as the parent individuals, where this binary mask decides which parent's value will be copied to the offspring individual - if the mask bit is 1, parent 1's value is copied to the first offspring individual; if the mask bit is 0, parent 2's value will be copied and vice versa with the second offspring individual. This then results into offspring individuals with a mixture of the values from both parent individuals determined by the binary mask. As there's a chance for duplicate values to be copied, this issue is resolved in the no_constraint part of our GA, which makes sure that there are no repeated values in our individuals.

Below is an example of an individual before and after the uniform crossover.

Parents

P1	1	4	3	6	9	5	2	8	10	7	1
P2	1	10	7	4	5	8	2	6	3	9	1

Offsprings

O1	1	4	7	4	5	5	2	8	10	9	1
O2	1	10	3	6	9	8	2	6	3	7	1

Using this binary mask

0	1	0	0	0	1	1	1	1	0	0

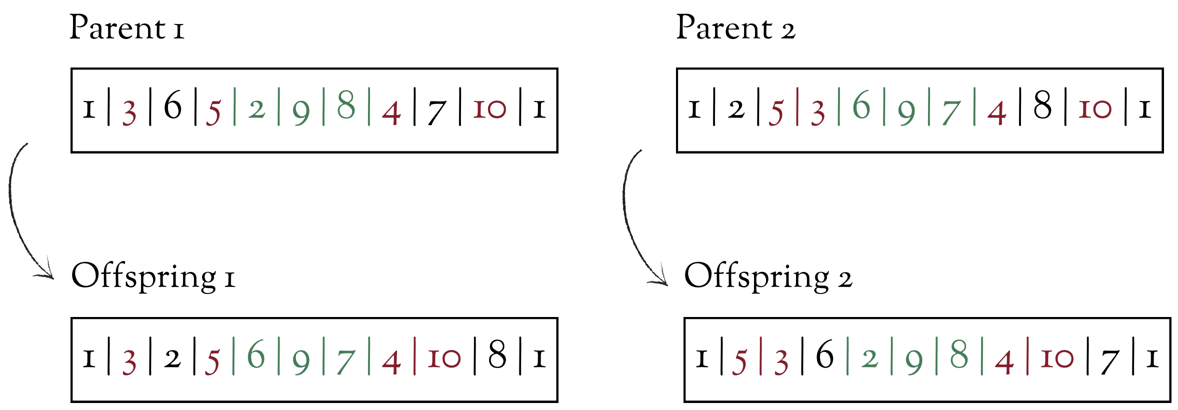
PARTIALLY MAPPED CROSSOVER

The principle behind Partially Mapped Crossover (PMX) is that it should preserve the arragement of genes in a parent and also allow variation of genes.

This operator starts by randomly defining two cutting points which divide the parents into a left, right and middle section. To create the offspring, the middle section is copied and the remaining positions are filled with the values of the corresponding positions from the other parent .

Since there's a high chance of duplicate values, our function handles the duplicates by following the mapping until an apropriate, non-duplicate position is found.

Below there's an example of PMX .

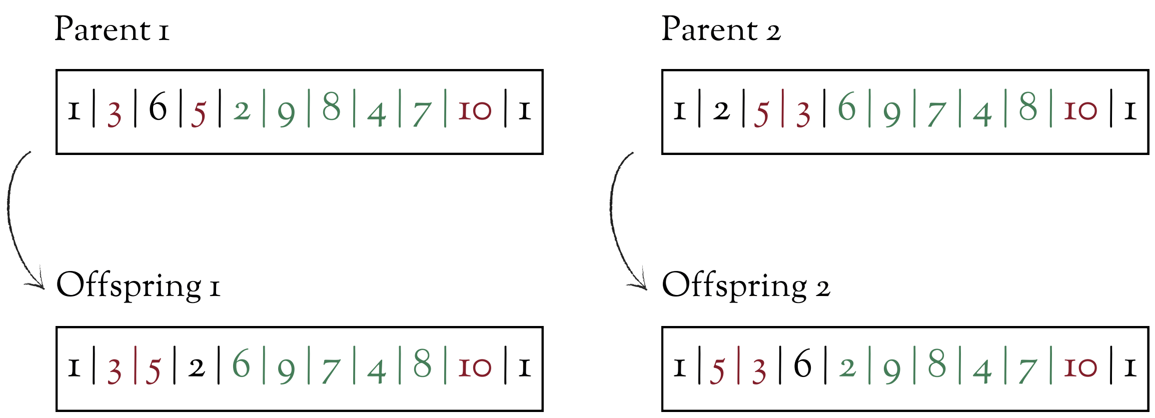


ORDER CROSSOVER

The Order Crossover (OX1) is a slight variation of the PMX with a different repairing procedure.

Similar to PMX, the OX1 starts by randomly defining two cuting points that divide the parents into three segments. To make the offsprings, the middle segment is copied from one parent to the corresponding positions. The other positions are then filled by considering the sequence of values in the other parent; starting right after the second cutting point and wrapping around to the beginning, if necessary.

Below is an example of the OX1.

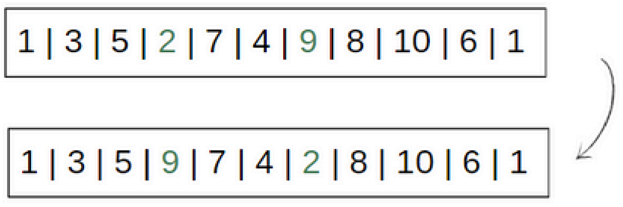


MUTATORS [4][7]

Mutation operators play a significant role in genetic algorithms, much like the natural process of genetic mutation found in biological species. These operators modify an individual's genetic composition in a controlled manner, similar to the random modifications that occur during natural genetic mutation. All the following mutation operators create variety by rearranging portions of an individual's genetic material, which might result in completely different solutions in the search space, consequently enhancing population variety.

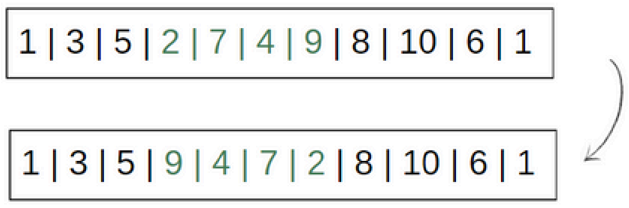
SWAP MUTATION

The swap mutation is a fundamental mutator characterized by its simplicity. It operates by randomly selecting two points within an individual and swapping the values at those points.
Below is an example of an individual before and after a swap mutation:



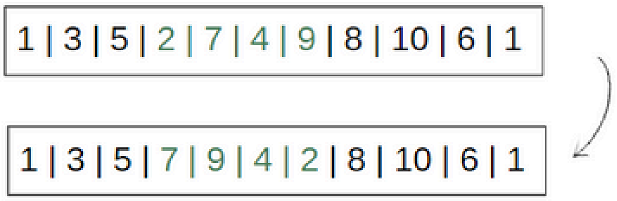
INVERSION MUTATION

The inversion mutation is a genetic operator that entails identifying two positions within an individual and reversing the order of the items between those positions.
Below is an example of an individual before and after an inversion mutation:



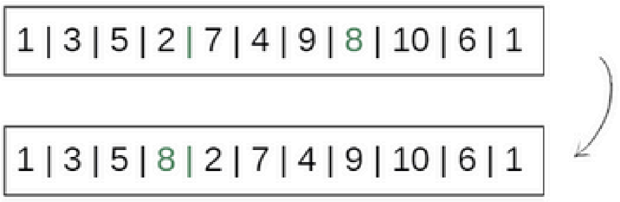
SCRAMBLE MUTATION

Unlike inversion mutations, which reverse a segment of an individual's genetic material, scramble mutations shuffle a segment of the individual's genetic material at random. It chooses two points within the individual and randomizes the sequence of the components between those positions.
Below is an example of an individual before and after a scramble mutation:



INSERTION MUTATION

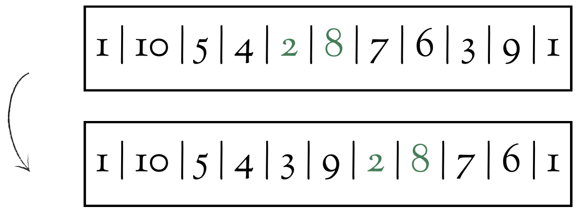
Insertion mutation involves removing a random element from the individual and putting it into another random location within the same individual.
Below is an example of an individual before and after an insertion mutation:



Gene to be removed: 8
Insertion Position: Before index 4

DISPLACEMENT MUTATION

Displacement mutation involves removing a segment from an individual and displacing it into a different location within the same individual.
Below is an example of an individual before and after a displacement mutation:



Segment to be displaced: 2 | 8
Displacement Position: Between genes 9 and 7

Methodology

In this section, the technological strategy that we undertook, to fulfill our project goals, is described. Our methodology includes all of the steps we took to create appropriate conditions for our genetic algorithm, such as dealing with the creation of individuals, populations, test data, genetic operators and, of course, the genetic algorithm, as well as a gridsearch to find the best hyperparameters and thus, the best solution. All of these phases were implemented utilizing the Python programming language and some of its accessible libraries.

Finally, for organizational purposes, we divided our code into several files and folders:

- a base folder including an individuals, a population and a data Python file;
- an operators folder including the Python files related to the various selectors, mutators, and crossovers;
- an algorithm folder and its utils (functions used to support it);
- a log folder, where the results of a generation run of our GA can be found (after running with your desired matrix, check the results there);
- a gridsearch Python file;
- a main Python file where the genetic algorithm can be run with the data to be inserted.

BASE

1. Data

As previously stated, we used several files to generate test data for individuals and populations.
Starting with our strategy of creating a matrix to represent the gain values for each game route, we developed the generate_geo_matrix function, that creates an example dataset (with values for each area between -300 and 900), which allowed us to confirm that if a random dataset was fed into the algorithm, it would function properly.
We also took into consideration the fact that a specific condition for Geo gains from Greenpath to Forgotten Crossroads was to be expected, therefore we made sure that we had that condition present in our test data, by setting the Geo gain from G to FC to be 3.2% less than the minimum of all other positive Geo gains.

2. Individuals

To later determine the most effective route, we began by developing a function that generated random individuals. These individuals consist of a list of 11 values, ranging from 1 to 11, where each value represents the area in the game:

– 1: Dirtmouth (D)
– 2: Forgotten Crossroads (FC)
– 3: Greenpath (G)
– 4: Queen's Station (QS)
– 5: Queen's Gardens (QG)
– 6: City Storerooms (CS)
– 7: King's Station (KS)
– 8: Resting Grounds (RG)
– 9: Distant Village (DV)
– 10: Stag Nest (SN)

To ensure that all routes begin and stop in Dirtmouth, the create_individuals() function shuffles regions between 2 (FC) and 10 (SN), followed by the addition of area 1 (D) at the beginning and the end. Since a number of preferences and specific constraints also needed to be taken into consideration, we then developed a function called no_constraints that checks whether the created individuals are valid according to the satisfaction of constraints that are to be imposed into them – ensuring that with each route taken, CS cannot be not visited after QG; RC is reached in the last half of the session; making sure that each session visits all of the areas once and only once – returning True if there are no constraints and False otherwise. Therefore, with these two functions, we could guaratee that only valid individuals were created.

In order to compare individuals, we created a function which computes the gain of a route taken, based on a certain data matrix (since we didn't have the actual gain values of each step, we used a random matrix obtained with the methods mentioned in the data section). It accesses the indexes in the matrix related to the areas of the individual and sums the gains of each step until reaching the final area.
Furthermore, if the QS-DV sequence is present, since our friend believes that visiting KS is unnecessary, we compare the route gains made if KS is retained and if it is skipped (replaced by a placeholder "0").

If the QS-DV sequence exists in our individual and a placeholder is introduced, it is discarded while computing the route gain. This means that if our current area is a placeholder, we use the area before it and the next area becomes the one following it. For example, if 507 is a segment of the individual at hand, the gain function would consider the sum of the 5 to 7 step. After that, the gains with and without a placeholder are compared and the greatest value is returned. If the highest is the one with a placeholder, we alter the actual individual's position for KS to a '0' . If there is no placeholder or the gains with one are lower, the individual remains unchanged and its route gain is returned.

Two additional functions were created to aid in the genetic operators phase, pre_operations and post_operations, which remove and insert the first and last areas of the route (D), allowing these operators to change all values in the route except for the start and end points.

Moreover, regarding our placeholder's logic, we created a function that fixes it when it is no longer required. That is, if the QS-DV sequence is no longer present after genetic changes, KS is put back into the placeholder's location. This possible condition subsequent to applying genetic operators is why we employed this logic rather than simply removing KS from an individual.

Lastly, for a better understanding of the algorithm's final route, we created a function that switches the area numbers to their respective initials. This provides an easier interpretation of the final route to suggest.

3. Population

In this file, merely two functions were created: one that creates a population of individuals of a certain size and another that calculates the population gain, which returns a list of gain values corresponding to each individual in the population.

OPERATORS

Selectors , Crossovers and Mutators

These files contain the seletion, crossover, and mutation methods stated in the background. In terms of the crossover and mutation operators, pre and post operations were used to ensure that the genetic alterations only affected values except for the start and finish points, as previously described. They also return the altered individual with a fixed placeholder if necessary.

THE ALGORITHM

After implementing our previous code, we arrived at the critical point of developing our algorithm. Our genetic algorithm (GA) reflects nature's evolutionary process. It begins by creating an initial population of potential solutions to our optimization problem. Each solution is evaluated for fitness.

With each individual assigned its unique gain value, we utilize one of five different selection operators (roulette/ranking/tournament/exponential rank/linear rank selection). On top of that, we ensured that the chosen parents were two distinct individuals. Given our optimization objective of maximizing Geo gains, these operators focus on individuals that have the largest gains.

The chosen individuals now act as parents and the next stage was to produce offsprings. To accomplish this, one of four different crossover operators (cycle/partially mapped/order/uniform crossover) and one of five different mutation operators(swap/inversion/scramble/insertion/displacement mutation) are applied to our parent individuals and if they remain valid and adhere to the constraints post-crossover and mutation, they are added to the population. This way we guarantee that all the individuals being used in the algorithm are still valid after any kind of alteration.

Once the offspring individuals rejoin the main population, our focus shifts to elitism, favoring individuals with the highest gains. In this process, priority is given to the best individual from the current population, preserving it in subsequent generations without alterations. The functions used to help in this process are found in the utils file.
To maintain consistency across different runs of our code, we introduced a seed hyperparameter for the random number generator, which ensures reproducibility of random processes, under a controlled choice of seed.

The above-mentioned algorithm will execute for the number of generations specified by the n_gens hyperparameter. To record our findings from each generation run, we use a verbose option that prints the results and a log_path hyperparameter that saves the data to a csv file in the log folder. With the results obtained, which are the gains of the populations for each generation, a line plot can then be made which graphically represents the best individuals and their corresponding fitnesses of each generation run, using the matplotib visualization library.

Finally after the algorithm is run, it returns the best individual found with its corresponding highest fitness.

GRIDSEARCH

With the aim of attaining the best possible solution, a key goal was to create a grid search algorithm that would be utilized to discover the most optimal combination of hyperparameters for this problem.
Two functions were used to accomplish this: the run_algorithm function, which executes the algorithm with specific parameters and measures performance by recording the best fitness value and runtime, and the grid_search function, which iterates over all possible parameter combinations and runs each combination multiple times using a multiprocessing pool to leverage multiple CPU cores for parallel execution. This provides statistical significance and improves efficiency. It gathers and averages the fitness values and runtimes for each combination before determining and returning the optimal parameter configuration. A Geo matrix is created for use in the parameter settings and the grid_search function is ran using the genetic algorithm, trying each combination 15 times with different parameters, as shown in the conclusion of this report.

Results

Following 15 rounds of our gridsearch and examining every possible combination of parameters, the following were found to generate an individual with the best fitness: A total of 15 generations , each having a population of 150. The Roulette Selection was chosen with the best mutation operator being the Displacement Mutation and the best crossover operator being the Cycle Crossover. The probability of crossover was 0.8, whereas the probability of mutation was 0.1. Finally, elitism was set to False and the random seed set to 12.

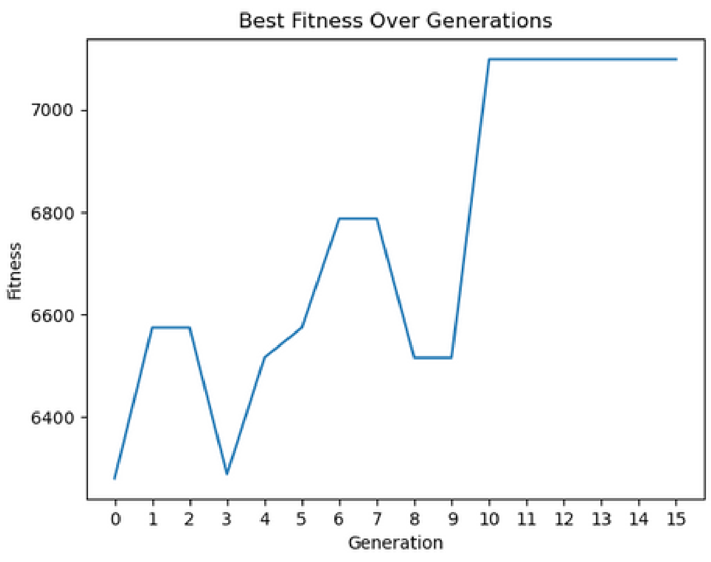


Fig.1 - Best Fitness per Generation

The plot above depicts the greatest fitness value attained during one of the algorithm's iterations, using the gridsearch parameters.

Summing up, what we can conclude is that the best route found for our friend to take was a route starting in Dirtmouth, followed by Distant Village, City Storerooms, Greenpath, Queen's Gardens, King Station, Forgotten Crossroads, Stag Nest, Resting Grounds, Queen's Station and finally going back to Dirtmouth.
With this route, a total of around 7098.8 Geo gains was obtained.

Best individual:
[1, 9, 6, 3, 5, 7, 2, 10, 8, 4, 1]

Best solution:
['D', 'DV', 'CS', 'G', 'QG', 'KS', 'FC', 'SN', 'RG', 'QS', 'D'] route.

Respective gain:
7098.800000000001

Conclusion

To sum up this project, we can agree that implementing genetic algorithms to identify the optimal route for the video game ‘Hollow Knight’ was an enourmous asset.

The implementation of this genetic algorithm involved creating a diverse population of potential solutions and refining them through iterative selection processes. We applied various genetic operators and conducted a thorough grid search to determine the best parameter settings.

Our project highlights the effectiveness of genetic algorithms in adressing complex optimization problems within a gaming environment. This work not only contributes for the field of video game optimization but also provides a versatile framework that can be adapted for other optimization challenges. The results of our genetic algorithm demonstrate its ability to enhance gameplay strategies and improve resource management in games.

Furthermore, this project demonstrates the synergy between computational intelligence and game strategy, showing how genetic algorithms can be effectively implemented to solve optimization problems.

References

[1] Vanneschi, L., & Silva, S. (2023). *Lectures on Intelligent Systems. In Springer eBooks.*

[2] Miller B. L., Goldberg D. E. *Genetic Algorithms, Tournament Selection, and the Effects of Noise. Complex Systems 9, 193- 212 (1995).*

[3] Pandey, H. M. (2016). *Performance Evaluation of Selection Methods of Genetic Algorithm and Network Security Concerns. Procedia Computer Science 78, 13-18 (2016).*

[4] Vanneschi L. (2024). *Genetic Algorithms. NOVA IMS, Universidade Nova de Lisboa.*

[5] Pandey, H. M., Deepti Mehrotra, Anupriya Shukla. (2015). *Comparative Review of Selection Techniques in Genetic Algorithm.*

[6] Hussain, A., Muhammad, Y. S., Sajid, M., Hussain, I., Shoukry, A. M., & Gani, S. (2017). *Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator. Computational Intelligence and Neuroscience, 2017, 1–7.*

[7] Nitasha Soni, Dr Tapas Kumar. (2014). *Study of Various Mutation Operators in Genetic Algorithms. (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 5 (3) , 2014, 4519-4521*

Annexes

Annex 1: Gridsearch Parameters

```
grid_search(ga, 15,

{’initializer’: [create_population],

’gain_matrix’: [data],

’evaluator’: [calculate_population_gain],

’selector’: [roulette_selection_max, ranking_selection_max,
tournament_selection_max,exponential_rank_selection,linear_rank_selection],

’crossover’: [cycle_crossover,pmx_crossover,uniform_crossover,ox1_crossover],

’mutator’: [swap_mutation, inversion_mutation, scramble_mutation, insertion_mutation,
displacement_mutation],

’pop_size’: [50,100,150],

’n_gens’: [5,15,20],

’p_xo’: [0.8, 0.9],

’p_m’: [0.2, 0.1],

’elite_func’: [get_elite_max],

’verbose’: [False],

’maximization’: [True],

’log_path’: [None],

’elitism’: [True,False],

’seed’: [12],

’fit_plot’: [None]})
```

Annex 2: Gain matrix and parameters of final algorithm

```
data = [[0, 506.1, -298.1, 63.7, 845.6, 879.4, 794.1, 117.8, 645.6, -183.7],
        [93.0, 0, 351.1, -264.0, -239.0, 649.8, 658.5, 471.8,-137.8, 860.8],
        [-256.5, 7.9, 0, 197.8, 741.5, 697.5, 23.8, -113.8, -285.8, 862.5],
        [703.1, 121.4, 387.4, 0, -9.3, 322.5, 8.7, 135.5, 251.0, 668.8],
        [864.5, 762.7, -95.3, 509.2, 0, -194.7, 572.9, 330.8, 8.2, -167.3],
        [385.9, 106.8, 860.1, 481.0, -271.4, 0, 322.9, -23.9, 262.1, 64.5],
        [217.0, 760.2, 647.8, -36.8, 474.7, 565.2, 0, 17.6, -201.2, 135.1],
        [723.8, 551.0, 851.7, 600.0, 62.5, 125.8, 18.8, 0, 98.6, 556.4],
        [729.6, 63.5, 609.1, 662.3, 613.7, 830.0, 400.0, 463.2, 0, 784.3],
        [-107.5, 621.5, 676.0, 123.7, -164.9, 892.2, 766.1, 524.6, 165.2, 0]]
```

```
ga(create_population,
  data,
  calculate_population_gain,
  roullete_selection_max,
  cycle_crossover,
  displacement_mutation,
  150,
  15,
  0.8,
  0.1,
  get_elite_max ,
  False,
  True,
  'log/test_log.csv',
  False,
  12,
  True)
```