NOVA IMS Information Management School

Final Project

Optimization Algorithms

Patrícia Bezerra: 20221907
Rita Silva: 20221920
Vasco Capão: 20221906

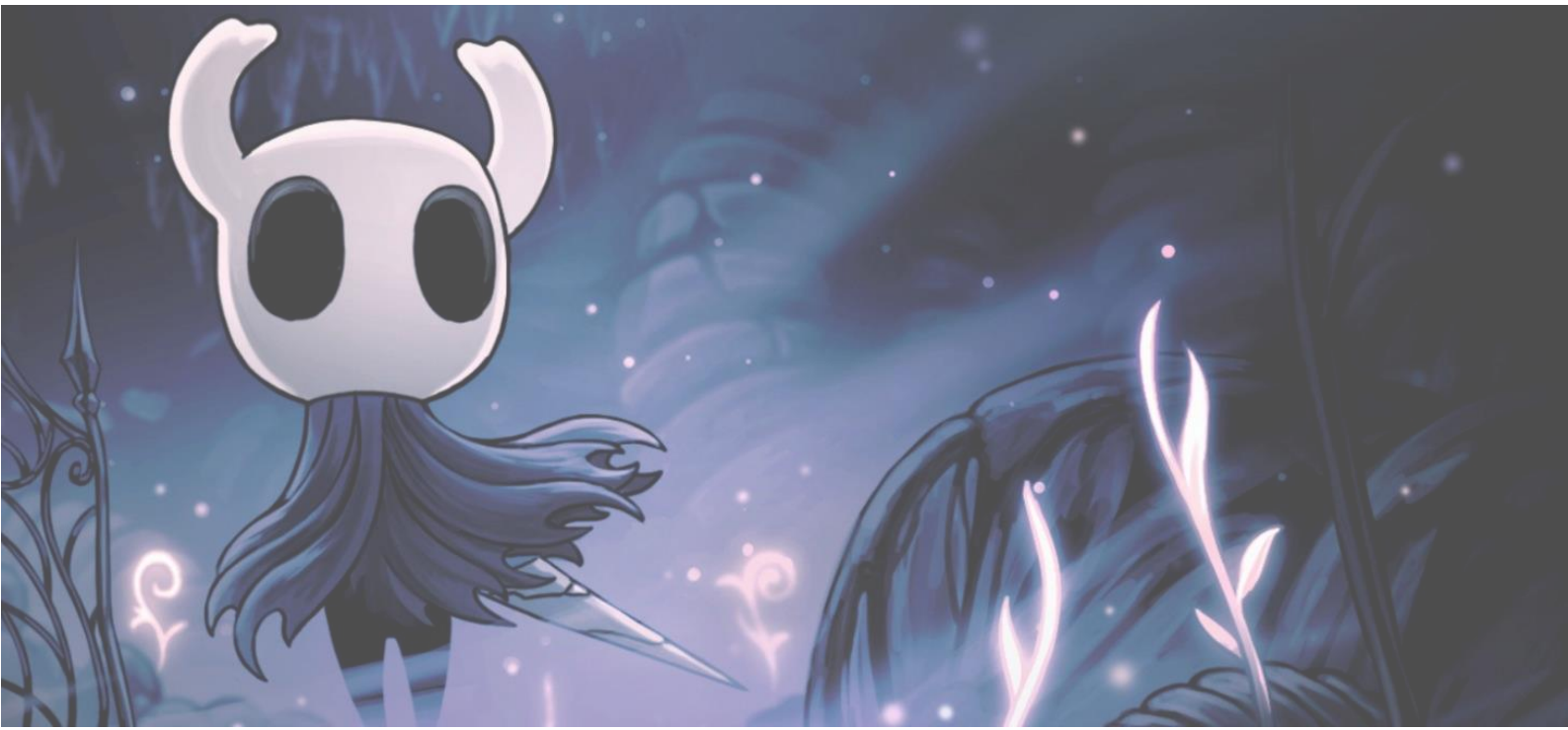

Table of Contents

Introduction

Genetic Algorithms (GAs) have proven to be powerful tools for solving complex optimization problems through the simulation of natural selection processes. This project builds upon the foundational implementation of GAs developed during the semester, with the goal of enhancing its effectiveness by exploring and implementing various genetic operators, specifically different types of crossovers and mutation. The objective is to apply these advanced techniques to a specific Optimization Problem (OP) in a gaming context, demonstrating both technical skill and creativity.

The given OP is inspired by the video game "Hollow Knight," where the player's objective is to maximize their Geo (in-game currency) earnings during a single session. The game involves navigating through various areas, each presenting different potential Geo earnings and risks of Geo loss. The challenge is to find an optimal route that maximizes Geo earnings while mitigating the risks of loss, under the constraint that the route must start and end in Dirtmouth, passing through all game areas at least once.

This project requires adapting the current GA implementation to handle the specific representations and constraints of the OP. The lack of provided data necessitates designing a robust pipeline that can effectively learn and adapt to varying Geo earnings and losses between game areas. The performance of the final model will be evaluated based on its ability to optimize the route compared to other models. By researching and integrating diverse crossover and mutation operators, this project aims to enhance the GA's capability to navigate the search space efficiently.

Genetic Algorithms

In order to find optimal solutions in a vast search space, Genetic Algorithms were utilized to address the presented problem. These algorithms leverage randomly generated solutions and iteratively improve them through processes such as selection, variation (including crossovers and mutations), and survival selection. Drawing on Darwin's theory of evolution, GAs mimic natural selection, reproduction, inheritance, variation, and competition. Selection algorithms operate based on fitness, while genetic operators modify an individual's structure regardless of its fitness level.

Upon reading the project description, it became evident that our problem bore similarities to the Traveling Salesperson Problem (TSP), prompting us to do some research during the project development. The primary distinction between the TSP and our problem lies in their respective objectives: the TSP aims for minimization (finding the shortest possible route visiting each city and returning to the origin), whereas our objective revolves around maximization (seeking the route with the least Geo loss).

Individual

In Genetic Algorithms, an individual represents a potential solution to a given problem within a specific population. Each individual expresses a distinct gene or variable arrangement that is subject to evaluation and improvement. Its role in the evolutionary process is determined by its fitness, which is defined by its inherent characteristics.

In our current problem, each individual is depicted as a list denoting a particular route. When generating an individual using the function `generate_valid_individual(areas)`, specific rules must be followed:

- The route must begin and end in ‘Dirtmouth’. Due to specific considerations in other parts of the code, ‘Dirtmouth’ will not be considered during individual creation, being later incorporated (figure 1).

```
# Remove 'D' if it is in the list
individual = [element for element in areas if element != 'D']
```

Figure 1

- The City Storerooms (CS) cannot immediately follow Queen’s Garden’s (QG) (figure 2). If this rule is violated, the approach we implemented is to change ‘CS’ with the area after it if it is not the last area to be visited. Otherwise, we changed the index of ‘CS’ and ‘QG’.

```
# Swap 'CS' and the following area if 'QG' is just before 'CS'
if 'QG' in individual and 'CS' in individual:
    qg_index = individual.index('QG')
    cs_index = individual.index('CS')
    if qg_index + 1 == cs_index:
        if cs_index + 1 < len(individual):
            individual[cs_index], individual[cs_index + 1] = individual[cs_index + 1], individual[cs_index]
        else:
            individual[cs_index], individual[qg_index] = individual[qg_index], individual[cs_index]
```

Figure 2

- The Resting Grounds (RG) can only occur in the last half of the route (figure 3).

```
# Ensure 'RG' is in the second half of the individual
midpoint = len(individual) // 2
if 'RG' not in individual[midpoint:]:
    individual.remove('RG')
    individual.insert(random.randint(midpoint, len(individual) - 1), 'RG')
```

Figure 3

- If the route includes the Distant Village (DV) immediately after Queen's Station (QS), there is no problem skipping King's Station (KS) (figure 4). In our approach, we remove 'KS' (replacing it by '-') if 'DV' is immediately after 'QS'.

```
# Skipping 'KS' if 'DV' is immediately after 'QS'
if 'QS' in individual and 'DV' in individual and 'KS' in individual:
    qs_index = individual.index('QS')
    dv_index = individual.index('DV')
    ks_index = individual.index('KS')
    if dv_index == qs_index + 1:
        individual.remove('KS')
        individual.insert(ks_index, '-')
```

Figure 4

During the project's execution, it became apparent that an evaluation function was necessary to verify the validity of the individuals. The function `is_individual_valid(individual)` encompasses all aforementioned conditions, returning 'False' if any condition is violated, and 'True' otherwise.

Considering that the problem was presented with 10 areas and 4 rules, our code was designed to focus on these rules. If more areas are added later, the code will still work and it will be possible to access those areas. However, we do not have restrictions that make it mandatory to visit those areas.

As previously mentioned, our algorithm requires the assessment of each individual's fitness. Consequently, we devised the function `evaluate_individual_geo(geo_matrix)` with the aim of maximizing fitness to mitigate Geo loss. To ensure accurate fitness evaluation, 'Dirtmouth' is appended to both the beginning and end of each individual's route within this function (figure 5).

```
# Add 'D' at the beginning and end of the individual.
individual_with_d = ['D'] + [str(area) for area in individual] + ['D']
```

Figure 5

Subsequently, we iterate through the locations, computing fitness and returning the calculated value if the individual is deemed valid. In the event of invalidity, a fitness value of '-99999999' is returned to drastically lower the likelihood of selection (figure 6).

```
# Calculate total Geo gained
for i in range(len(individual_with_d) - 1):
    from_area = individual_with_d[i]
    to_area = individual_with_d[i + 1]

    # If the origin or destination area is not in the matrix, assign 0 to the Geo value
    if from_area not in geo_matrix.index or to_area not in geo_matrix.columns:
        total_geo += 0
    else:
        total_geo += geo_matrix.loc[from_area, to_area]

# if individual is valid, return the total_geo, otherwise, -99999999
return total_geo if is_individual_valid(individual_with_d) else -99999999
```

Figure 6

Population

Similar to how nature explores possibilities through variation, we initiate our search by establishing a diverse pool of potential solutions. This initial population size is a crucial first step. Each solution, represented by an "individual," embodies a unique arrangement relevant to the problem at hand.

By iteratively generating individuals, we gradually build our population to a desired size. In the code, we created the function `generate_pop(pop_size)`, where 'pop_size' is an integer, representing the size of the population. This process ensures a rich pool of possibilities, encompassing a variety of solution approaches. This diverse foundation is critical for the next stage, where the algorithm refines these solutions through selection and modification, ultimately leading us closer to the optimal outcome.

```
def generate_population(geo_matrix):
    new *
    def generate_pop(pop_size):
        areas = list(geo_matrix.keys())
        return [generate_valid_individual(areas) for individual_from_pop in range(pop_size)]
    return generate_pop
```

Figure 7 - Generate a population of individuals for a Genetic Algorithm based on geo.

To achieve this diversity, we employ a random generation process. Each individual acts as a candidate answer, with its quality assessed through a fitness evaluation. This evaluation measures how well an individual performs, allowing us to distinguish promising candidates from less effective ones. So, we have also created a function named, `pop_evaluation(population)` that will calculate the fitness of each individual in the population.

```
def evaluate_population_geo(geo_matrix):
    new *
    def pop_evaluation(population):
        return [evaluate_individual_geo(geo_matrix)(individual) for individual in population]
    return pop_evaluation
```

Figure 8 - Evaluate the fitness of each individual in a population based on geo.

Selection Algorithms

Tournament Selection

By employing Tournament Selection, we can potentially reduce the computational burden by assessing the fitness of only a subset of individuals rather than the entire population for selecting a single candidate. Whenever there's a need to pick an individual, a set of *k* individuals is randomly selected from the current population with a uniform distribution. This process occurs each time an individual has to be selected. Following this, the individual selected is the one demonstrating the highest fitness level among the *k* randomly chosen individuals from the population and will enter the parent´s population. Defining the tournament size directly influences the selection pressure of the algorithm - a smaller *k* implies lower selection pressure, whereas a larger *k* indicates higher selection pressure.

This selection algorithm is divided into two parts:

1. The process of selecting the *k* individuals for tournament evaluation is entirely independent of their fitness, thereby guaranteeing the randomness of the solution.
2. The selection of the top individuals for inclusion in the parent population is a deterministic stage, meaning the outcome is predictable and consistent given the same input. Moreover, it heavily relies on the fitness evaluations of the various *k* individuals.

To iterate the process for selecting the next individual, it's crucial to remember that the entire algorithm must be repeated, including the generation of a fresh pool of *k* individuals randomly selected from the current population. This ensures that each execution of the algorithm remains entirely independent from the others.

Tournament selection is notable for its efficiency and simplicity in tuning selection pressure. By selecting individuals through random tournaments, this method can be more efficient than other algorithms by reducing the need to evaluate the fitness of every individual. We can also verify that by using this method to select an individual, we have a positive probability of choosing the worst individual in the population.

Fitness Proportionate/Roulette Wheel Selection Algorithm

The Fitness Proportionate Selection Algorithm involves choosing the most optimal individuals from a population based on their fitness values. Specifically, the likelihood of selecting a particular individual from the initial population is determined by the division of his fitness value by the sum of all the fitness values of the individuals in the population.

It is evident that the greater the fitness value the higher the probability of selecting the individual since the denominators are the same to all the individuals.

Why do we call this Roulette Wheel Selection Algorithm?

A straightforward way to simulate the algorithm is to envision a roulette wheel divided into *N* segments each representing an individual in the population. The area of each segment is directly proportional to the fitness value. To select an individual, simply "play" a roulette game by spinning the wheel and allowing a ball to roll. The individual corresponding to the segment where the ball lands will be selected. In other words, if the ball stops in segment sk, then *k* will be the chosen individual.

Ranking Selection

Ranking Selection initiates by arranging individuals in the population according to their fitness levels, usually from the lowest to the highest. Each individual is assigned a selection probability which is determined by the individual´s position in the ranking. For computing this probability, we utilize the position held by the individual in the ranking, divided by the total sum of ranking indices of all individuals.

There are some differences between the ranking selection algorithm and the roulette wheel and the main one is their sensitivity to fitness differences. So, in ranking selection, changes in an individual's fitness might not alter its position in the ranking, but they could significantly impact its selection probability in fitness proportionate selection.

Crossovers

Crossover, a genetic operator in evolutionary algorithms, mimics the biological process of sexual reproduction to generate offspring with a blend of parental characteristics.

By selecting segments of genetic material from two parent chromosomes and recombining them at a crossover point, new individuals are formed, inheriting traits from both parents. This recombination fosters genetic diversity within the population, enhancing the algorithm's capability to explore novel regions of the solution space and converge towards optimal solutions.

Unlike mutation, where the occurrence is typically rare, as is true in nature, there is typically a very high chance that a son will inherit his parent's characteristics. This increases the likelihood of crossover occurring. As such, crossover plays a crucial role in the genetic algorithm's quest for optimal solutions by facilitating the exploration of diverse genetic combinations.

Following some investigation, we selected crossover techniques that we believed would work well with our code and are frequently applied to the "Traveling Salesman Problem."

One Point Xover

Standard crossover in genetic algorithms operates by taking two individuals, parents, as input. It then randomly selects a crossover point, which is a position between two consecutive characters in the string representation of the individuals. This crossover point remains the same for both parents. Next, it exchanges the substrings located to the left and right of this crossover point between the two parents, thereby generating two new individuals, offspring.

(crossover point at position 4)
Parent 1: 1 2 3 4 **5 6 7 8 9**
Parent 2: 9 8 7 6 **5 4 3 2 1**

Offspring 1: 1 2 3 4 **5 4 3 2 1**
Offspring 2: 9 8 7 6 **5 6 7 8 9**

Two Point Xover

This is a particular instance of the One Point Crossover method. On each chromosome, two randomly selected points are selected, and genetic material is exchanged at these locations.

(crossover point at positions 2 and 5)
Parent 1: 1 2 **3 4 5** 6 7 8 9
Parent 2: 9 8 **7 6 5** 4 3 2 1

Offspring 1: 1 2 **7 6 5** 6 7 8 9
Offspring 2: 9 8 **3 4 5** 4 3 2 1

Partially mapped Crossover

Partially Mapped Crossover (PMX) begins by randomly selecting two crossover points on the parents' chromosomes. These crossover points divide the parents' chromosomes into three segments: left, middle, and right. The intermediate portion of the parents' chromosomes is copied directly to the offspring chromosomes. This ensures that certain genes remain unchanged in the offspring. For each offspring, the genes in the left and right segments that need to be replaced are identified. Additionally, a mapping relationship is established to determine the placement of these genes to avoid redundancy. This involves replacing the gene from the first parent with its corresponding gene from the second parent. Based on the established mapping relationship, repeated genes in each chromosome are found and replaced. This ensures that there is no duplication of genes in the offspring.

(crossover point at positions 2 and 5)
Parent 1: 1 2 **3 4 5** 6 7 8 9
Parent 2: 9 8 **7 6 5** 4 3 2 1

'Offspring 1': * * **7 6 5** * * * *
'Offspring 2': * * **3 4 5** * * * *

Gene Mapping: 7 <-> 3 ; 6 <-> 4 ; 5 <-> 5
Offspring 1: 1 2 **7 6 5 4 3** 8 9
Offspring 2: 9 8 **3 4 5 6 7** 2 1

Order Crossover

Order Crossover (OX) begins by randomly selecting two crossover points on the parents' chromosomes, and genetic material is exchanged at these locations. Then starting from parent 1, we make a list with the values of parent 1, starting at the second crossover point, and then we remove from this list the values that are in the middle of the crossover points of parent 2. Finally, we add the result of the previous step to offspring 1, in which the sequence will be placed starting with the second crossover. The process is repeated for offspring 2.

(crossover point at positions 2 and 5)
Parent 1: 1 2 **3 4 5** 6 7 8 9
Parent 2: 9 8 **7 6 5** 4 3 2 1

‘Offspring 1’: ** **7 6 5** ****
‘Offspring 2’: ** **3 4 5** ****

6 7 8 9 1 2 3 4 5 → 8 9 1 2 3 4 (remove 7 6 5)
4 3 2 1 9 8 7 6 5 → 2 1 9 8 7 6 (remove 3 4 5)

Offspring 1: 3 4 7 6 5 8 9 1 2
Offspring 2: 7 6 3 4 5 2 1 9 8

Mutators

Mutation algorithms in Genetic Algorithms introduce random changes to individuals' chromosomes, ensuring genetic diversity and preventing premature convergence. It entails randomly choosing chromosomes and modifying one or more of their genes. A mutation's chance of occurring is determined by a quantity called the mutation rate. Generally, it is desirable to define a low mutation rate because mutations are uncommon.

By randomly altering genetic material, mutation explores new regions of the search space, complementing crossover's role in combining existing solutions.

Swap Mutation

The swap mutation just involves randomly switching two components from an existing chromosome to create a new one.

Original Individual: 1 2 3 4 5 6
Mutated Individual: 1 **4** 3 **2** 5 6

Scramble Mutation

A scramble mutation, commonly employed in permutation representations, selectively chooses a subset of genes within the chromosome and randomly shuffles their values, thereby fostering genetic diversity without altering the subset's original position within the chromosome.

Original Individual: 1 2 3 4 5 6
Selected Subset: 1 | **2 3 4 5** | 6
Mutated Individual: 1 4 3 5 2 6

Inversion Mutation

In inversion mutation, a subset of genes is chosen similar to scramble mutation, yet the alteration occurs through a reversal of the sequence encompassed by two randomly selected points, offering a distinct mechanism for genetic diversification.

Original Individual: 1 2 3 4 5 6
Selected Subset: 1 | **2 3 4 5** | 6
Mutated Individual: 1 5 4 3 2 6

Displacement Mutation

Displacement mutation, a versatile operator in genetic algorithms, operates by randomly selecting a substring within the chromosome and relocating it to a different position outside the original substring, preserving the sequential order of the remaining elements, thereby fostering genetic diversity and exploration.

Original Individual: 1 2 3 4 5 6
Selected Subset: 1 2 | **3 4 5** | 6
Mutated Individual: **3 4 5** 1 2 6

Distance Based Mutation

In distance-based mutation (DMO), the genetic operator introduces variability by first identifying a random position within the chromosome. Subsequently, it identifies elements within a specified distance around this position. These selected elements undergo a shuffling process within the interval defined by the distance parameter. The mutation process is governed by two key variables: 'g', representing the randomly chosen position, and 'distance', indicating the range of neighbouring elements to be considered for shuffling.

g = 3 ; distance = 2
Original Individual: 1 2 3 4 5 6
Mutated Individual: 1 6 4 2 5 3

Center Inversion Mutation

In Center Inversion Mutation, the chromosome is partitioned into two segments by randomly selecting a split point. Subsequently, each segment undergoes a reversal operation, effectively inverting the sequence of genes within each part. Finally, the resulting segments are combined and transferred to the offspring.

Original Individual: 1 2 3 4 5 6
Splitting point: 1 2 3 4 | 5 6
Mutated Individual: 4 3 2 1 6 5

Utils

The `utils` folder includes several useful functions to enhance the functionality of our project.

The `get_n_elite_max()` function is designed to create a function that, when called, returns the best individual from a population based on its fitness value. This higher-order function returns an inner function, `get_elite`, which takes two lists: one representing the population of individuals and another containing the corresponding fitness values. By using `np.argmax(pop_fit)`, the function determines the index of the individual with the highest fitness value. It then returns the elite individual and its fitness value.

The `list_to_dataframe(list_of_lists, indexes)` function converts a list of lists into a DataFrame, using specified indexes to label both rows and columns. This function was created because we found it more beneficial to work with a DataFrame rather than a list of lists, as it simplifies the code and data manipulation.

The `generate_geo_dataframe(geo_names, negative_prob=0.2,seed=0)` function generates a non-symmetric DataFrame of geo values with specific rules. The resulting DataFrame has zeros on the main diagonal, predominantly positive values, and ensures that the gain from Greenpath (G) to Forgotten Crossroads (FC) is at least 3.2% less than the minimum positive gain, excluding the diagonal.

Algorithm

The core structure of Genetic Algorithms hardly changes, so we only made a minor adjustment to the code we developed in practical classes.

The algorithm begins by creating an initial population containing *n* individuals and evaluating its fitness. Subsequently, a loop is initiated and iterated through all generations: an empty list is created to store the offspring (the offspring population). Continuously, until this new list matches the size of the initial population, two individuals (parents) are selected from the population. Depending on the probability, crossover operators are applied, or the individuals are simply “reproduced” without alterations. Additionally, depending on the probability, mutation operators are applied to each offspring resulting from one of the aforementioned processes, with the resultant offspring being inserted into the offspring population. When both lists are of equal size, the offspring population becomes the new current population.

As the input parameter ‘elitism’ is set to true, the algorithm chooses the best individual from each generation to preserve it for the next, ensuring the retention of the most promising individual at each interaction.

Furthermore, by including the parameter ‘verbose’ and setting it to true, we can visualize the best individual and its fitness at the end of each generation.

The adjustment implemented pertains to the individuals within the final population, where we have added to the beginning and end of each individual ‘Dirtmouth’ (as explained previously in the report).

Hyperparameters	Description
initializer	Function that creates the population
evaluator	Function that computes the fitness of the population
selector	Function that selects the individuals from the initial population
crossover	Function that performs the crossover on two individuals (parents)
mutator	Function that performs mutation on an individual
pop_size	Size of the population
n_gens	Number of generations for the algorithm
p_xo	Probability of crossover occurring
p_m	Probability of mutation occurring
elite_func	Function to select the elite individual from the population
verbose	Boolean indicating if there should be information printed, or not, on each generation and final solution.
log_path	Path of the location of the CSV file to store the values at each generation
elitism	Boolean indicating if elitism occurs in the algorithm
seed	If it's different than None (default), set a random seed value so that the results can be replicated.

Table 1 – Hyperparameters used in the Genetic Algorithm function.

Grid Search

One popular technique for optimizing hyperparameters is Grid Search. To find the ideal combination, it searches through a preset subset of hyperparameters in an exhaustive search. The search is not unlimited since the values of these hyperparameters are restricted to a particular set. Within this set, all potential combinations are computed, and their respective performances are assessed using predetermined criteria.

This method is particularly advantageous because it automates the process of hyperparameter tuning, which would be impractical to perform manually. Grid Search makes sure that no alternative configuration is missed by methodically analysing all possible combinations. This increases the probability of finding the ideal set of hyperparameters for a particular model.

This systematic approach, coupled with the Python `itertools` library, allowed us to conduct an exhaustive search over a predefined set of hyperparameters.

In our implementation, we decided to divide the parameter range between two computers to analyze a larger interval of values. The parameters that we tested are the following:

- **Population Size and Number of Generations**: We varied the population sizes (pop_size = [20, 40, 50, 60, 80, 100]) and the number of generations (n_gens = [10, 20,30, 40, 50, 60]) to determine their impact on the GA's performance.

- **Elite Function**: We used an elite function (elite_func = get_n_elite_max( )) to ensure that the best individual from each generation is preserved in the next generation.

- **Selection, Crossover, and Mutation Operators**: We tested various operators to see which combinations yielded the best results:

    - **Selectors**: Tournament selection (tournament_selection_max(15) and tournament_selection_max(30)), fitness-probability selection (fit_probability_selection_max()), and ranking selection (ranking_selection_max()).

    - **Crossovers**: One-point crossover (one_point_xover), two-point crossover (two_point_xover), PMX crossover (pmx_crossover), and OX crossover (ox_crossover).

    - **Mutators**: Swap mutation (swap_mutation), scramble mutation (scramble_mutation), inversion mutation (inversion_mutation), displacement mutation (displacement_mutation), distance-based mutation (distance_based_mutation), and center inversion mutation (center_inversion_mutation).

- **Probabilities**: We explored different probabilities for crossover (p_xo_range = [0.70, 0.75, 0.80, 0.85, 0.90, 0.95]) and mutation (p_m_range = [0.30, 0.25, 0.20, 0.15, 0.10, 0.05]).

Furthermore, the use of 15 distinct seeds met the project specifications using 15 different matrices, playing a role in ensuring the reproducibility of the results.

Results and Discussion

During the project, we encountered some difficulties that slowed down our progress.

The first challenge was figuring out how to make sure that the individuals, after undergoing crossover and mutation, were still valid according to the rules in the project description. We considered two options:

1. Creating a function that would correct the individuals and then return them to the algorithm.

2. Assigning a very low fitness score of '-99999999' to any invalid individuals during evaluation.

We chose the second option. This way, the likelihood of selecting an invalid individual would be almost 0. If we had chosen the first option, we would have ended up modifying the individuals in the same way we did during their creation, which would undermine the natural application of crossover and mutation.

Additionally, we received the professor's clarification just as we were making the final adjustments to our code. Although we were aware of the issue, we ended up replacing King's Station (KS) with '-' if the player visited Distant Village (DV) right after Queen's Station (QS).

Regarding GridSearch, this was the most challenging part of the project for various reasons, and we spent the most time understanding and solving the issues.

The first problem we encountered was that storing the results in lists caused our computer memory to crash. We fixed this by storing all the results in a CSV file, allowing us to access and analyze the data later.

The second major problem, which took us several days to solve, was that our code was creating a list (an individual) inside another list. This became problematic during crossovers because the unexpected length would either cause an error or make GridSearch stop running. We resolved this by modifying our elitism function, which was returning a list with an individual. After this adjustment, our code ran without errors, and GridSearch was completed successfully.

Upon completing the GridSearch process, we obtained insightful results that shed light on the optimal parameters for our algorithm. We conducted two iterations of GridSearch, each revealing distinct sets of parameters that yielded the best fitness scores.

```
'''
RESULTS:

1º Parameters for GridSearch

                selector      crossover               mutator  p_xo  p_m  pop_s  gen  fitness_mean
3077  inner_tournament  pmx_crossover  displacement_mutation   0.9  0.3     50   30   7596.842667
3047  inner_tournament  pmx_crossover  displacement_mutation   0.8  0.3     40   30   7594.466667
3076  inner_tournament  pmx_crossover  displacement_mutation   0.9  0.3     50   20   7584.242667
2564  inner_tournament   ox_crossover  displacement_mutation   0.8  0.3     50   30   7583.066667
2534  inner_tournament   ox_crossover  displacement_mutation   0.7  0.3     40   30   7573.533333
-----------------------------------------------------------------------------------------------

2º Parameters for GridSearch
                selector        crossover               mutator  p_xo   p_m  pop_s  gen  fitness_mean
5471  inner_tournament  two_point_xover  displacement_mutation  0.85  0.15    100   60   7758.066667 -> Best parameters
4046  inner_tournament  one_point_xover  displacement_mutation  0.95  0.25     80   60   7743.200000
4044  inner_tournament  one_point_xover  displacement_mutation  0.95  0.25     80   40   7743.200000
4045  inner_tournament  one_point_xover  displacement_mutation  0.95  0.25     80   50   7743.200000
5498  inner_tournament  two_point_xover  displacement_mutation  0.95  0.15    100   60   7736.933333
'''
```

Figure 9 - GridSearch best results

In the first iteration, our analysis led us to identify the following parameters as optimal:

- Selector: Tournament Selection (*k*=15)
- Crossover: PMX Crossover
- Mutator: Displacement Mutation
- Crossover Probability (p_xo): 0.9
- Mutation Probability (p_m): 0.3
- Population Size (pop_s): 50
- Number of Generations (gen): 30

These parameter configurations resulted in a mean fitness score of approximately 7596.84.

In the second iteration of GridSearch, a different set of parameters emerged as the most favorable:

- Selector: Inner Tournament (*k*=30)
- Crossover: Two-Point Crossover
- Mutator: Displacement Mutation
- Crossover Probability (p_xo): 0.85
- Mutation Probability (p_m): 0.15
- Population Size (pop_s): 100
- Number of Generations (gen): 60

This parameter combination outperformed others, achieving a mean fitness score of 7758.07.

Given the bigger mean fitness score obtained with the parameters from the second iteration, we have selected this configuration as the optimal choice for our algorithm.

Conclusion

This project exemplifies the powerful application of Genetic Algorithms to complex optimization problems, particularly within a gaming context inspired by "Hollow Knight." By adapting the foundational GA implementation to address the specific constraints and representations of maximizing Geo earnings, we have demonstrated the effectiveness of GAs in navigating and optimizing intricate search spaces. The integration of diverse crossover and mutation operators significantly enhances the GA's performance, showcasing an improved ability to explore and exploit potential solutions efficiently.

Our results of GridSearch showed that the hyperparameters that lead to the best values of Geo earnings (on average) were:

- Selector: Inner Tournament (k=30)

- Crossover: Two-Point Crossover

- Mutator: Displacement Mutation

- Crossover Probability (p_xo): 0.85

- Mutation Probability (p_m): 0.15

- Population Size (pop_s): 100

- Number of Generations (gen): 60

In conclusion, our project serves as a compelling testament to the versatility and adaptability of Genetic Algorithms. Through systematic GridSearch, we have identified optimal hyperparameters (detailed above), resulting in significantly improved Geo earnings. Moreover, this project has served as an invaluable learning experience, deepening our understanding of Genetic Algorithms and their practical application in solving complex optimization problems.

References

1. *Genetic algorithms - mutation*. Tutorialspoint. (n.d.). https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm
2. Karazmoodeh, A. (2023, December 31). *Mutations in genetic algorithms*. LinkedIn. https://www.linkedin.com/pulse/mutations-genetic-algorithms-ali-karazmoodeh-u94pf/
3. Predicting the outcome of mutation in genetic algorithms. (n.d.). https://www.fep.up.pt/docentes/fontes/FCTEGE2008/Publicacoes/B1.pdf
4. Redalyc.combined mutation operators of genetic algorithm (n.d.-b). https://www.redalyc.org/pdf/2652/265219635002.pdf
5. (PDF) genetic algorithms for the travelling salesman problem: A review of representations and operators. (n.d.-a). https://www.researchgate.net/publication/226665831_Genetic_Algorithms_for_the_Travelling_Salesman_Problem_A_Review_of_Representations_and_Operators
6. Analyzing the performance of mutation operators (n.d.-a). https://arxiv.org/pdf/1203.3099

7. Kora, P., & Yadlapalli, P. (2017, march). Crossover Operators in Genetic Algorithms: A Review. ResearchGate. https://www.researchgate.net/publication/315175882_Crossover_Operators_in_Genetic_Algorithms_A_Review

8. Dou, Xin-Ai & Yang, Qiang & Gao, Xu-Dong & Lu, Zhen-Yu & Zhang, Jun. (2023). A Comparative Study on Crossover Operators of Genetic Algorithm for Traveling Salesman Problem. 1-8. 10.1109/ICACI58115.2023.10146181.