STIGLER'S DIET PROBLEM:

GENETIC ALGORITHM APPROACH

Group 32:
Pedro Peças, 20220586
Tiago Figueiredo, 20220495
Aidar Zaripov, 20220663
Lizaveta Barisyonak, 20220667

Division of labor:
Full genetic representation of the problem, Selection/Mutation/Crossover, Next Generation, Solver - Pedro Peças and Tiago Figueiredo. Algorithms comparison, Fitness, Code review, Report - Aidar Zaripov and Lizaveta Barisyonak

Introduction

Stigler's diet problem is a well-known optimization problem aiming to find the cheapest combination of food items that meet specific nutritional requirements. In this project our objective is to apply genetic algorithms to Stigler's diet problem and evaluate their efficacy in finding cost-efficient food combinations. This report presents our methodology, experimental setup, results, and analysis. The code and the report itself can be found here: https://github.com/AfonsoVip/Stigler_Diet_CI40_Group_32.git

Genetic representation

The detailed description of the problem can be found here https://developers.google.com/optimization/lp/stigler_diet. In short, there are 77 food items with certain price and nutrition values of each item. Also there are 9 constraints (minimal requirements) on the nutrients (such as calories, proteins, calcium etc.).

In Stigler's diet problem, the genetic representation of the solution involves encoding the quantities of food items selected for the optimal diet. Each individual in the genetic algorithm population represents a potential solution, and the genetic representation typically uses a binary or integer encoding scheme.

One common approach is to use a binary string of fixed length, where each bit corresponds to whether a particular food item is included or excluded from the diet. For example, a '1' may indicate the presence of an item, while a '0' represents its absence. The length of the binary string corresponds to the total number of food items considered in the problem.

Alternatively, an integer encoding scheme can be used, where each gene value represents the quantity or proportion of a specific food item in the diet. The integer values can be discretized to represent feasible quantities, such as the number of servings or grams of each food item.

In this project we decided to use the binary representation approach mostly for its simplicity (easy to interpret and implement, allows easy manipulation during crossover and mutation processes) and computational efficiency (reduced memory consumption and computational complexity).

Fitness function

The fitness function in Stigler's diet problem evaluates the desirability of a particular solution (i.e., a binary string representing the selection or exclusion of food items in the

diet). The fitness function aims to measure how well the solution meets the specified nutritional requirements while minimizing the cost. The fitness function consists of two components:

1. Cost Component: This component calculates the total cost associated with the selected food items in the diet. It multiplies the binary representation of food items with their respective prices and sums up these costs to determine the total cost of the diet.
2. Nutritional Component: This component assesses the extent to which the solution meets the required nutritional constraints. It computes the nutritional values (e.g., proteins, carbohydrates, fats, vitamins, minerals) of the selected food items based on their binary representation and compares them to the specified minimum nutritional requirements. A penalty in the form of multiplying the fitness function by 10 and adding 99 is applied for not meeting each nutrient requirement. This penalty will heavily penalize the solutions which do not meet the minimal requirements, including all zeros solutions.

Other fitness functions can be tested to observe their impact on the genetic algorithm, however in this project we didn't do that as the one described above was already able to provide satisfactory results.

Selection function

Two selection techniques were implemented in this project, represented by the following functions in the code:

a. ***selection***: this function selects two parent individuals for the genetic algorithm based on their fitness values. It chooses the fittest individuals from the first four shuffled members of the population.

b. ***tournament_selection***: this function selects two parent individuals from the population by running a **k-way tournament selection**. It picks k random individuals and chooses the two individuals with the lowest fitness values. In this project k = 4.

Crossover function

Two crossover techniques were implemented in this project, represented by the following functions in the code:

a. ***crossover***: This function performs a **single-point crossover** on the given parents, producing two offspring. The crossover point is selected randomly between the first and last indices.

b. **two_point_crossover**: This function performs a **two-point crossover** on the given parents, producing two offspring. The crossover points are selected randomly between the first and last indices.

The crossover rate we have chosen for the final configuration is 0.7. The crossover rate determines the likelihood of applying crossover to generate offspring.

Mutation function

Two mutation techniques were implemented in this project, represented by the following functions in the code:

a. ***mutate***: This function performs **mutation** for a list of individuals. The mutation probability is determined by *mutation_rate*. The mutation changes the value of a randomly picked index in the `food_selections` list.

b. ***swap_mutation*** and ***mutate_v2***: These functions perform a **swap mutation** on a single individual and a list of individuals, respectively. The function randomly chooses two different indices of the `food_selections` list and swaps their values.

Mutation rate we have chosen for the final configuration is 0.1.

Elitism

Elitism was implemented in the project The inclusion of elitism helps to preserve the best solutions found so far from one generation to the next, preventing the loss of good solutions due to crossover and mutation. However, iit is important to find a balance with elitism, as too high of an elitism rate can hinder diversity and exploration in the population, potentially leading to premature convergence to suboptimal solutions. We have chosen the **elitism rate to be equal to 10%**. Which means that the top 10% of individuals (based on their fitness values) will be selected as elite individuals and directly copied to the next generation. 10% is a typical value for this kind of problem and provides a reasonable trade-off between exploration and exploitation.

Final Parameters

Configuration we have chosen for the final implementation is the following one:

initial_population_size = 50
generations = 500
crossover_rate = 0.7
mutation_rate = 0.1
elitism_rate = 0.1

Each run of the genetic algorithm we start with 50 randomly selected solutions. The selection process continues for 500 generations. Other parameters were discussed above. The parameters were tuned manually and found to work well together, yielding good results in terms of execution time and convergence to the good results. To determine the "best" configuration, one could try multiple combinations of these parameters and compare the output fitness values and convergence speeds. However, in this project the provided configuration already demonstrated a good balance in terms of exploration and exploitation of the search space.

Comparison of different operators

Different operators can affect the convergence of the genetic algorithm. In the provided code, two types of crossover functions are implemented: single-point and two-point crossover. Two types of mutation functions are presented: mutate (flip) and swap mutation. Also two types of crossover function are used. Different operators can result in varying convergence speeds, diversity within the population, and the algorithm's ability to find optimal solutions. It is important to test different operators to see how they affect the algorithm's performance for a particular problem. The results in terms of total cost (value of the fitness function) obtained through running the algorithm **20 times** are presented in the table below.

	Selection function	**Mutation function**	**Crossover function**	**Mean total cost**
1	*selection*	*mutate*	*crossover*	7.75
2	*selection*	*mutate*	*two_point_crossover*	7.13
3	*selection*	*swap_mutation*	*crossover*	6.33
4	*selection*	*swap_mutation*	*two_point_crossover*	6.74
5	*tornament selection*	*mutate*	*crossover*	8.62
6	*tournament selection*	*mutate*	*two_point_crossover*	7.18

7	*tournament selection*	*swap_mutation*	*crossover*	8.84
8	*tournament selection*	*swap_mutation*	*two_point_crossover*	7.31

Overall, it can be seen that the configuration with simple selection function, swap mutation and simple crossover perform slightly better on this type of task. However the difference between the different configurations is relatively small and we think any of them can be used to solve the Stigler's diet problem.

Potential improvements

The provided implementation provides good results for the diet optimization problem as it finds a cost-efficient diet that meets the minimum nutrient requirements. However further possible improvements could include:

1. Implementing and testing more crossover and mutation techniques to assess their impact on the GA's performance.

2. Testing different parameter configurations such as number of generations, initial population size, elitism rate, mutation rate and crossover rate to find an optimal balance for exploration and exploitation.

3. Making the implementation more abstract and applicable to various optimization problems, both minimization and maximization. The current implementation is focused on the specific problem of minimizing the total cost of a diet while meeting nutrient requirements. If desired, the implementation can be made more abstract and work for both minimization and maximization problems by modifying the fitness function to suit the particular optimization problem being worked on.

4. Incorporating constraints or additional criteria to the fitness function for more complex scenarios, such as food preferences, allergies, dietary restrictions, or regional availability.

5. Trying to experiment with different genetic representations of Stigler's diet problem. Current representation doesn't allow you to choose the fractions of each product, it only allows you to choose the products as a whole as the representation is binary.

Overall, the current implementation in this project demonstrates a successful application of a genetic algorithm to the Stigle's Diet optimization problem and can serve as a foundation for further improvements and adaptations.