Stigler’s Diet Problem

COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION 22/23

Group 15: Afonso Lavado 20220631, Clara Santos 20220637, Mª Margarida Graça 20220602, Nuno Dias 20220603
NOVA IMS |

Distribution of labor: Afonso Lavado 25%, Clara Santos 25%, Mª Margarida Graça 25%, Nuno Dias 25%

Git repositor link: https://github.com/20220631/CIFO_Project_15

1. Introduction

For our project, we chose the Stigler's Diet Problem. This problem aims to minimize the overall cost while satisfying specific nutritional daily requirements. The information on the nutritional content and cost of our list of foods is in a python file called 'data.py'. Our data is the original with 77 different foods and corresponding information and the original constraints.

To address this problem, we have opted for a representation where the quantity of a specific food item is represented by a decimal float number. We didn't choose to use binary or integer representation because the nutritional quantities of our products are big, so if we choose them, it will fill all the nutritional requirements. Instead, we used a representation that can take values between 0 and 1 with a step size of 0.01. Our choice allows us to explore a wider range of possible solutions, considering different amounts of food items to achieve the desired nutrient requirements.

We aim to evolve populations of individuals that converge to an optimal solution and return a list of foods that will be the cheapest possible combination while fulfilling the nutritional requirements needed.

2. Genetic Algorithms

a) Fitness Function

For our fitness function we start with an initial fitness of zero and a list 'nutrients_totals', so we can add each nutrient that the products have. We run a for loop for each product in the individual adding the price to the fitness and adding the nutrients to the 'nutrients_totals'. Then we have a penalty, that basically is checking if the nutrients minimums are being met, and if they aren't, it applies the penalty to the fitness, returning in the end the fitness plus the penalty.

We decided to use elitism since it can help maintain the best solution found so far and prevent it from being lost in subsequent generations.

b) Selection

For this section of our project, we used Fitness Proportionate Selection, Tournament Selection and Rank Selection. Since all these algorithms were lectured in class, we will not explain how they work.

We noticed the FPS is not a good fit for our problem since it's only based on fitness values. This can lead to a biased selection towards individuals with lower costs but potentially inadequate nutrient fulfilment. It's also relevant to mention that in tournament selection we used size = 10 since it allows diversity by giving weaker individuals a chance, while still providing enough pressure to favor fitter individuals.

c) Crossover

Crossover is essential in any genetic algorithm since it creates an individual that has traces of both parents combined, giving to the population diversity and new potential better solutions to the given problem. We use single-point crossover, multi-point crossover, uniform crossover and partially matched crossover. In this section, we will explain the multi-point and uniform crossovers.

Multi-point crossover works like the single point crossover seen in class, but we can give an input ‘num_points’ that will be the number of points that will be crossed. It starts by checking if the ‘num_points’ is less than the length of the parent, then it generates random ‘num_points’ crossover points that then are sorted in ascending order. It creates two empty lists ‘offspring1’ and ‘offspring2’ so it can store the solutions. The function then loops over ‘num_points’ + 1 times. In each iteration, it determines the start and end points for the current segment to be swapped. For the first segment, the start is 0 and the end is the first crossover point. For the last segment, the start is the last crossover point, and the end is the length of the parent solution. For all other segments, the start and end are the current and next crossover points. The function then checks if the current segment number is even or odd. If it's even, it adds the segment from p1 to offspring1 and the segment from p2 to offspring2. If it's odd, it does the opposite: it adds the segment from p2 to offspring1 and the segment from p1 to offspring2. And then returns the two offspring.

Uniform crossover is a method where each corresponding gene is independently chosen from the parents with an equal probability of being inherited by the offspring. In our implementation, we use randint(0, 1) to randomly choose between 0 and 1. When the value is 0, the gene from the first parent is selected and appended to the first offspring, while the gene from the second parent is appended to the second offspring. Conversely, when the value is 1, the opposite occurs: the gene from the first parent is passed to the second offspring, and the gene from the second parent is assigned to the first offspring.

d) Mutation

Mutation introduces diversity into the population by randomly modifying the genetic material of individuals. It helps prevent premature convergence by maintaining a certain level of exploration. For our project, we use swap mutation, creep mutation, uniform mutation and random resetting mutation. Uniform and random resetting mutations are explained below. The code of these mutations is in the ‘mutation.py’ file.

Creep mutation selects a random gene from the individual and then randomly chooses between -0.05 and 0.05, to add to that gene by the end it checks if the gene doesn't break the limits of our ‘valid_set’, returning the mutated individual.

Uniform mutation selects an individual and then, for each of its genes, it generates a random integer number between 0 and 1 with a step size of 0.01 to change the gene. For each gene, this mutation only happens if a random number between 0 and 1 is smaller than a predefined mutation rate, in our case 0.5. At the end of this loop, the mutated individual is returned.

Random resetting mutation randomly selects a gene from an individual of the population and in that selected gene, it changes the value to one between 0 and 1 with a step size of 0.01. It then returns the mutated individual. It only makes this change for one gene.

3. Testing All Methods

In our ‘1 Average Fitness Per Combination.py’ file in the ‘Plots’ folder, we created a code to test all the combinations from selection, mutation and crossover. We plotted the bar chart below by making 4 for loops, 1 for mutation, 1 for selection, 1 for crossover and 1 for the number of iterations of each combination. The bar chart has the average fitness value for each possible combination (36 combinations in total, 10 rounds each, with a population size of 50 and 40 generations). Each one of the combinations generates different values, so we can say that the operators have influence in the convergence of the genetic algorithms.

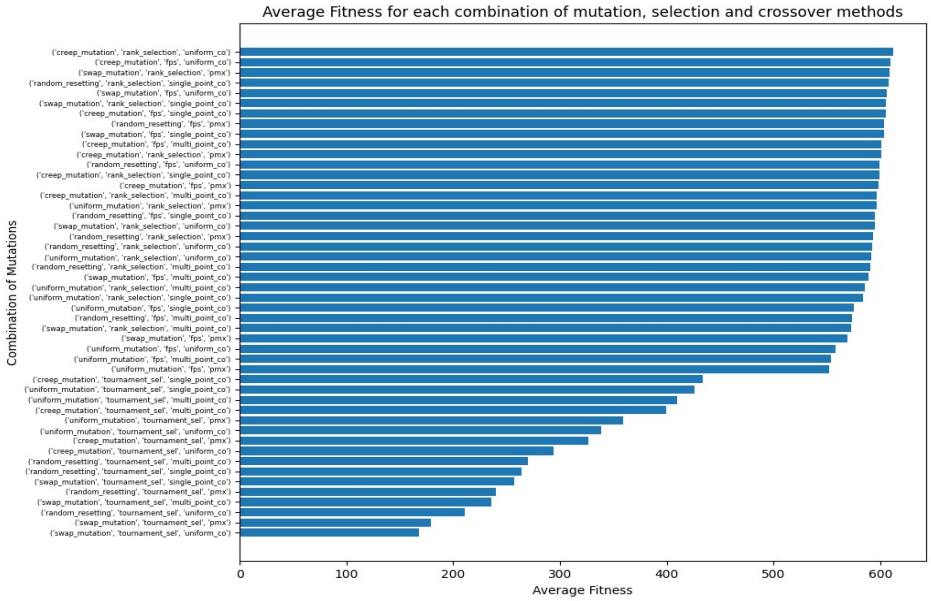

Figure 1 - Bar Chart

Since our problem is a minimization one, we can observe that there are methods of each section (selection, crossover and mutation) that are similarly good. Therefore, we will analyze the top 4 combinations in a more profound manner by plotting box plots. Looking at the boxplots below, the best combination is tournament selection with swap mutation and uniform crossover. This code is in the ‘2 Average Fitness of Top 4 Combinations.py’ in ‘Plots’ folder.

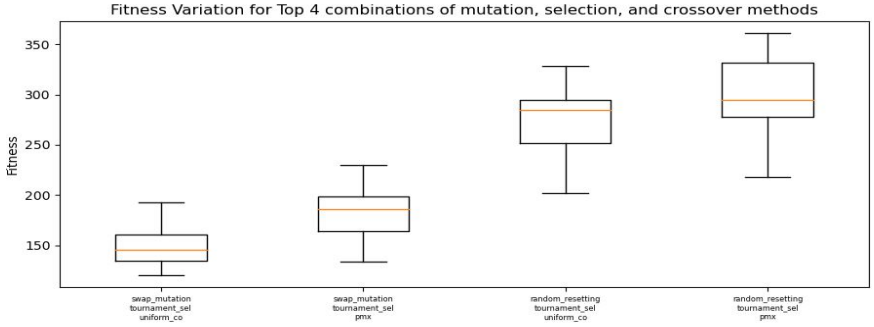

Figure 2 - Boxplot

4. Best Model

To test our best model, we needed to determine the best number for population and generations. For that, we plotted the two figures below. For both cases, we ran 5 times for each iteration and calculated the average fitness values among each generation and population size, respectively. The grey area shows the minimum and maximum fitness in every possible value for the search parameters. We conclude that the best number of generations is 113 and the population size is 91, where the fitness is lower.

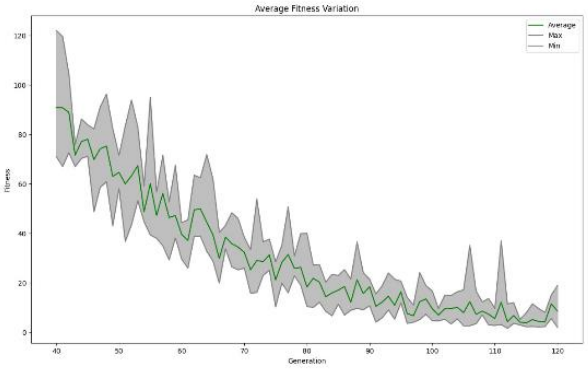

Figure 3 - N of generations per fitness

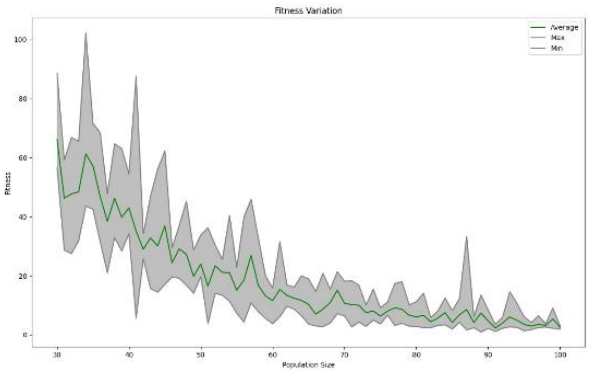

Figure 4 - Population size per fitness

Our best fitness value was 1.068. In this problem, this means that by spending 1.068 cents per day (3.8982 dollars per year, considering it has 365 days), you can purchase the following list of foods while paying the least amount of money to fulfill the nutritional requirements. The list is composed of 6 products, that are: White Bread (Enriched), Unit: 1 lb., 1939 price (cents): 7.9, Quantity: 0.01; Apples, Unit: 1 lb., 1939 price (cents): 4.4, Quantity: 0.02; Cabbage, Unit: 1 lb., 1939 price (cents): 3.7, Quantity: 0.03; Onions, Unit: 1 lb., 1939 price (cents): 3.6, Quantity: 0.06; Sweet Potatoes, Unit: 1 lb., 1939 price (cents): 5.1, Quantity: 0.02; Navy Beans, Dried, Unit: 1 lb., 1939 price (cents): 5.9, Quantity: 0.08.

Simplifying this list and converting the weights to grams, it resumes to: White Bread – 4.536 g, Apples – 9.072 g, Cabbage – 13.608 g, Onions – 27.216 g, Sweet Potatoes – 9.072 g, Navy Beans, Dried – 36.287 g. The nutritional values for this solution are shown in the table below.

Commodity	Quantities	Calories (kcal)	Protein (g)	Calcium (g)	Iron (mg)	Vitamin A (KIU)	Thiamine (mg)	Riboflavin (mg)	Niacin (mg)	Ascorbic Acid (mg)
White Bread (Enriched)	0,01	15	488	2,5	115	0	13,8	8,5	126	0
Apples	0,02	5,8	27	0,5	36	7,3	3,6	2,7	5	544
Cabbage	0,03	2,6	125	4	36	7,2	9	4,5	26	5369
Onions	0,06	5,8	166	3,8	59	16,6	4,7	5,9	21	1184
Sweet Potatoes	0,02	9,6	138	2,7	54	290,7	8,4	5,4	83	1912
Navy Beans, Dried	0,08	26,9	1691	11,4	792	0	38,4	24,6	217	0
Minimum Values		3	70	0,8	12	5	1,8	2,7	18	75
Totals		3,036	157,17	1,349	70,93	7,172	4,002	2,704	22,42	281,23

Table 1 - Nutritional values

This solution was reached with 500 runs and the parameters chosen above (number of generations 113 and the population size 91) and a range for valid set from 0 to 1 with a step size of 0.01.

We are happy with the result because we improved the solution found by Stigler (in 1393), using modern computational methods of optimization. We decreased the cost from 39.93 dollars per year to 3.8982 which is a big improvement on the solution.

There are still improvements that can be made, in terms of problem and code. In terms of datasets, improvements can be made by updating the prices for today's economy or updating the foods, as well as the nutritional requirements, to more recent studies. In terms of our code, we could have done combinations of all the possible values of population, number of generations, and valid set parameters. We could change the step size of valid set to 0.001, increase the number of runs, add restrictions to the problem so the solution can have more variety of products or create restrictions on specific food allergies or preferences.

We tried to improve by increasing the values for our parameters, with number of generations equal to 200, population size equal to 200, and 2000 runs. This gives us a best fitness value of 0.754 (2.75 dollars per year), having 4 products. However, this is not ideal since it takes a lot of computational power and time and the improvement shown is not that big. The nutritional values for this solution are shown in the table below.

Commodity	Quantities	Calories (kcal)	Protein (g)	Calcium (g)	Iron (mg)	Vitamin A (KIU)	Thiamine (mg)	Riboflavin (mg)	Niacin (mg)	Ascorbic Acid (mg)
Corn Meal	0,01	36	897	1,7	99	30,9	17,4	7,9	106	0
Cabbage	0,01	2,6	125	4	36	7,2	9	4,5	26	5369
Spinach	0,01	1,1	106	0	138	918,4	5,7	13,8	33	2755
Navy Beans, Dried	0,1	26,9	1691	11,4	792	0	38,4	24,6	217	0
Minimum Values		3	70	0,8	12	5	1,8	2,7	18	75
Totals		3,087	180,38	1,197	81,93	9,565	4,161	2,722	23,35	81,24