Group Number:

Group 30

Github: link

Optimizing Stigler's Diet: A Genetic Algorithm Approach for Nutritional Efficiency

Members:

20220617 | Nichita Zamisnii

20220584 | Ricardo Almeida

20220620 | Ianis Rușitoru

Statement of Contribution: In this project, our aim was to ensure an equitable distribution of contributions among all members of the group. We advocated for continuous monitoring and information sharing, holding regular meetings. Initially, the project was collaboratively developed to facilitate problem filtering and establish a clear approach. Subsequently, tasks were allocated. Ianis assumed responsibility for ensuring the quality of the individual and population classes, developing and formalizing genetic algorithms such as CRP, and implementing minimization for FPS. Additionally, Ianis conducted data preprocessing and contributed to the report. Nichita focused on developing genetic algorithms, including scramble mutation and ranking selection, and also contributed to the report. Ricardo contributed to the project by developing genetic algorithms like inversion_mutation_crp and pmxo, the evaluate_hyperparams and visualization functions, and provided support for the report.

I. Introduction

The problem we have chosen to undertake is the Stigler's diet problem, a well known optimization task. The primary objective of this project was to demonstrate the effectiveness and efficiency of genetic algorithms in solving the knapsack type problem. To achieve this goal we have used the resource linked in the project guidelines. As a nutritional goal we have kept the standard daily nutritional requirements as defined in Stigler's problem in 1939.

Through comprehensive experimentation and analysis, we have identified optimal or near-optimal diet plans that minimize costs while meeting nutritional needs. We achieved this by implementing various selection, crossover and mutation operators. Additionally, we have discussed the strengths and limitations of the genetic algorithm approach and provided insights into potential areas for further improvement.

II. Preprocessing

We have copied the data from the provided source and have performed various transformations to be able to use it with the implementation of *charles.py.* Among these transformations, we have created a list which contains just the nutrient information for each food item, as well as creating a list just for the food item prices (fitness). We have also multiplied the daily recommended nutrient intake (constraints) to reflect the yearly goal. Finally we have reduced the number of foods using the concepts of Pareto efficiency.

III. Representation

The first representation we have used was a bit string of length 77, each bit representing a food item from the original dataset. We chose an integer representation because we desired to be able to select food items multiple times. Throughout the development of the project, our representation of the problem has evolved as we acquired more knowledge. For instance, we have identified the Pareto efficient foods in terms of nutrients and cost, which narrowed the length to 27. This development did not provide significant improvements in terms of fitness, but it made the algorithm execution considerably faster by reducing the calculation volume by 65%.

IV. Fitness

Fitness function modifications

The *get_fitness* function developed in class assumed a knapsack problem using a binary representation with a single constraint. For our use case we had to modify the function to accommodate both our representation and increased number of constraints. To calculate fitness, we first initialize two lists with value 0, *fitness* (price) and *current_nutrients*. Following, the price of food items gets multiplied by the corresponding representation (amount) and is added to *fitness*. Then the nutritional information is multiplied by the respective amounts in the *current_nutrients*. Finally if the nutritional needs are not met fitness is penalized.

V. Mutation

After creating the fitness function, we have realized that we need to develop a mechanism to increment/decrement the amounts in the representation since we are using integer encoding. To achieve this, we couldn't use binary mutation and opted to develop mutation functions that could fit our needs.

Controlled Random Perturbation

We have created *controlled_random_perturbation* (CRP) which with probability *mut_prob* would multiply all items in the representation, with different values between 0.9 and 1.1. Thus a decrement scenario involves multiplying the current bit value with a random continuous number between 0.9 and 1. For incrementation, with a random continuous number between 1 and 1.1. We have identified that it performs better when values are off-set and centered in 1 so the change is proportionate and nondestructive. We have tweaked these values and noticed that it converges faster for large changes (0.7 to 1.3 for instance). On the other hand, using smaller steps (0.99 to 1.01) considerably decreases the speed of convergence but the fitness is consistently better due to not overshooting the optimal values as much. We thought that this mutation can perform good because the random choice gives evolution a chance to do smaller or bigger steps within that range when appropriate to improve fitness.

Whenever this mutation occurs all the bits are changed, but the magnitude is different for each bit due the randomness. We have tested this mutation with bitwise probability of perturbation (**Annex Figures 1 and 2**) to allow some bits to remain unchanged, but the results did not vary much; it slightly hindered/improved some of the combinations.

Inversion Mutation

We have implemented the *inversion_mutation_crp* which is an inversion mutation variant coupled together with the previously mentioned CRP. We thought of introducing this type of mutation to explore new potential nutritional combinations. By taking segments of genotype and inverting their order, it was an attempt to preserve essential food combinations and maintain the nutrition diversity of the solutions. The results achieved were satisfactory but it did not do better than the simple *controlled_random_perturbation* in this particular problem.

Scramble Mutation

The function scramble_mutation_crp was implemented after exploring the literature and understanding that this type of mutation allows handling complex optimization problems. It has the property of navigating through complex landscapes while still maintaining diversity, and allows the best individuals to achieve better fitness by simply shuffling randomly the genotype. The use of this mutation converges to good results with the CRP effect, but even with this development, simple *controlled_random_perturbation* performed better.

VI. Selection

Concerning the selection procedure, we have used several methods, namely tournament_sel, *fps*, and *ranking_selection*. In the final assessment we decided to use solely tournament selection as we understood from the literature that it is the most used technique for selection in genetic algorithms. The reason is that it is easy to implement, computationally efficient, and achieves the best results overall. Moreover, the other two selection methods were many times slower to run, making them unfeasible to execute for many generations across a statistically significant amount of runs. We have implemented them in the code nonetheless. Regarding fitness proportionate selection**,** we needed to implement the minimization option, which we achieved by inverting fitness so that the lower fitness is the more likely to get picked. This selection method was very slow and didn't perform well given that all fitness values were close together. We ought to solve this particular problem by implementing **ranking selection** as this method uses ranks instead of proportionate slices to fitness on the roulette wheel spin. This selection proved a bit faster but results were still not satisfactory compared to tournament selection. All these selection methods work for both maximization and minimization problems.

VII. Crossover

The crossover techniques used were *singlepoint_crossover*, *pmxo,* and *arithmetic_xo*. We did not need to adjust any of the functions except *pmx* (from class) since we encountered indexing errors. In the class implementation we encountered errors when executing the code because in *pmx_offspring*, the code attempts to find the index of an element (temp) in the parent using y.index(x[y.index(temp)]). This approach can lead to incorrect indices and conflicts if the elements are not found in the expected positions.

In the implementation we provided, the indexing operations are handled differently. Instead of using nested index operations, we use a while loop to find the correct index by repeatedly searching for the element in the parent and updating the index accordingly (while child1[i] is None: i = parent2.index(parent1[i])). This ensures that the correct index is found, even if the element is not in the expected position.

VIII. Configuration and Experimental Setup

We have used the knapsack implementation provided in class with no additional modifications. We use a high mutation rate and a high crossover rate. From theory we know that usually crossover should have a high probability and mutation a low probability. In our case the algorithm converged faster and to better solutions while maintaining a high mutation rate. This is because our main engine for the optimization are the increments/decrements of the bit values after the initialization which can only be achieved through the developed mutations. We use a valid range of 0 to 50 to initialize every food item quantity. We have chosen this interval to optimize for speed, based on practical experimentation. We have noted that initializing at 0 takes longer to converge and achieves similar results. We have executed *pop.evolve* without elitism, but it seems to perform better when we use it. The comparison can be viewed in **Annex Figures 3 and 4**.

IX. Results and Discussion

To evaluate each implementation, we have constructed a function that works as a grid search trying every combination of the operators mentioned before. In total there are 9 different combinations to benchmark since we use only *tournament_sel* (size 2) for the aforementioned reasons. To store the best individual at each generation, we have slightly modified the *Population* class from *Charles* to be able to insert every best individual from each generation into an array. Every combination has been executed 60 times for 1000 generations with population size 100 using a mutation rate of 0.95 and a crossover rate of 0.8. Results per generation for every combination across runs are averaged to ensure reproducible solutions.

In **Figure 1** we have plotted the results of the execution. It is clear that the combination of *tournamen_sel*, *controlled_random_perturbation* and *single_point_co* performs best in all criteria, it converges the fastest and to the best results. A common trait of all the top 4 combinations is that they use either *crontrol_random_perturbation*, *single_point_co* or both. *Scramble_mutation_crp* performed well except when combined when *arithmetic_xo*. Combinations that used *inversion_mutation_crp* performed the worst overall as can be observed from the gaps in fitness in the graph. The exact fitness value of these combinations at every 100 generations can be found in the **Annex Figure 5**.

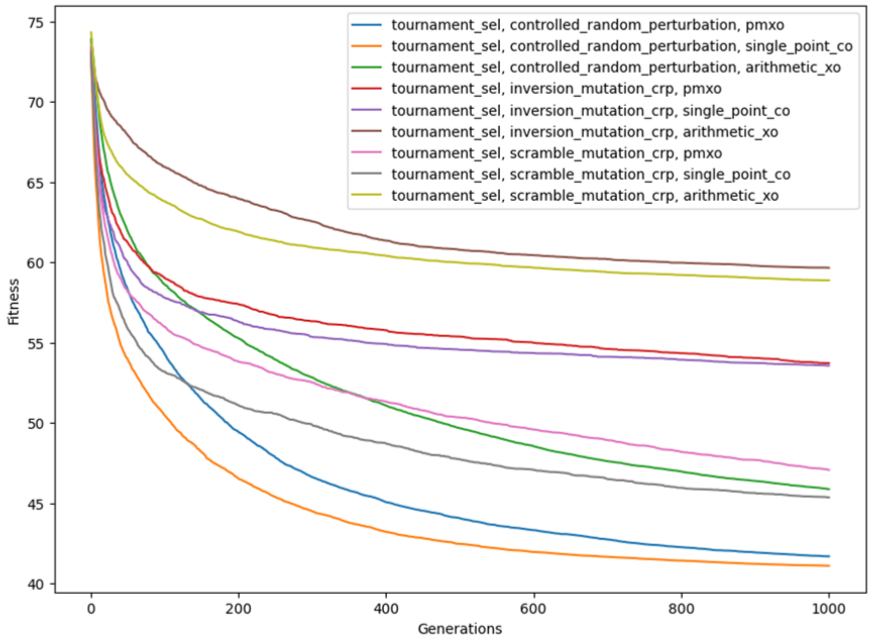


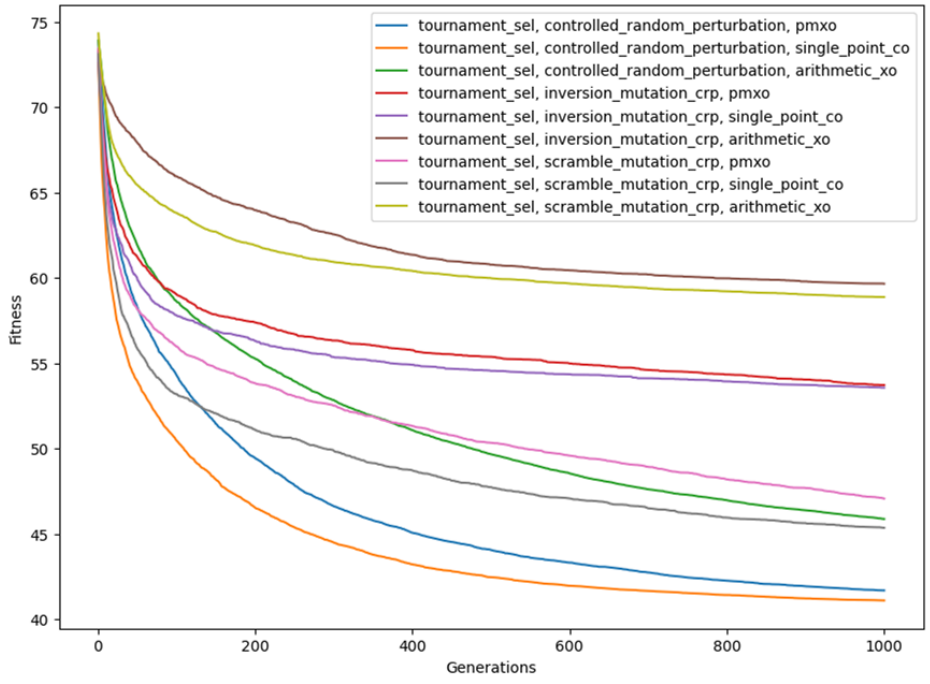
Figure 1 - Results of genetic algorithms on Stigler's Diet problem

The best fitness ever achieved across runs was 40.49$. It met the yearly nutritional goals with the following excesses: 32 kcal, 28936g protein, 2.41g calcium, 17768mg Iron, 61 (KIU) vit A, 887mg Vit B1, 2.56mg Vit B2, 3569mg Niacin, 298mg Vit C. The list of foods and amounts can be found in **Annex Figures 6 and 7**. For this project we used the standard nutritional goals as defined by Stigler to have a way to benchmark our solution against the results he achieved. Using linear programming in 1939 he achieved a final price of 39.97$/year with a similar main food list and excess amounts. In the implementation provided in the assignment description a solution of 39.66$ is achieved. We consider our result a close enough approximation given the number of generations executed. This being said we have to acknowledge there is still a quantifiable potential for improvement in how we select and mutate the individuals. The algorithm we implemented uses the 27 Pareto foods, some in disproportionately small amounts. Meanwhile we noted that in the assignment resource, despite using all the 77 foods the algorithm can precisely identify a subset of foods, typically 3-6 depending on the requirements and achieves better results just by optimizing those quantities.

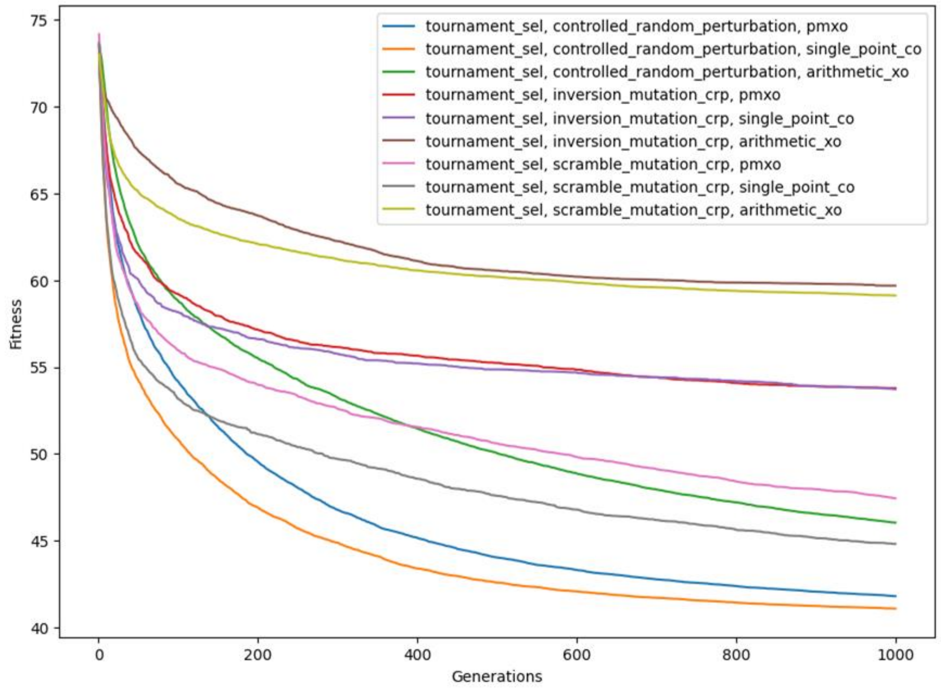
Annexes

Figures 1 and 2: Standard CRP vs CRP with 0.95 Probability

Comparison of the standard CRP (**every bit** changes by a random amount in the range 0.9-1.1, with no exception) with the bitwise probability of mutation implementation (every bit has a 95% to be incremented/decremented by a random amount in the range 0.9-1.1).

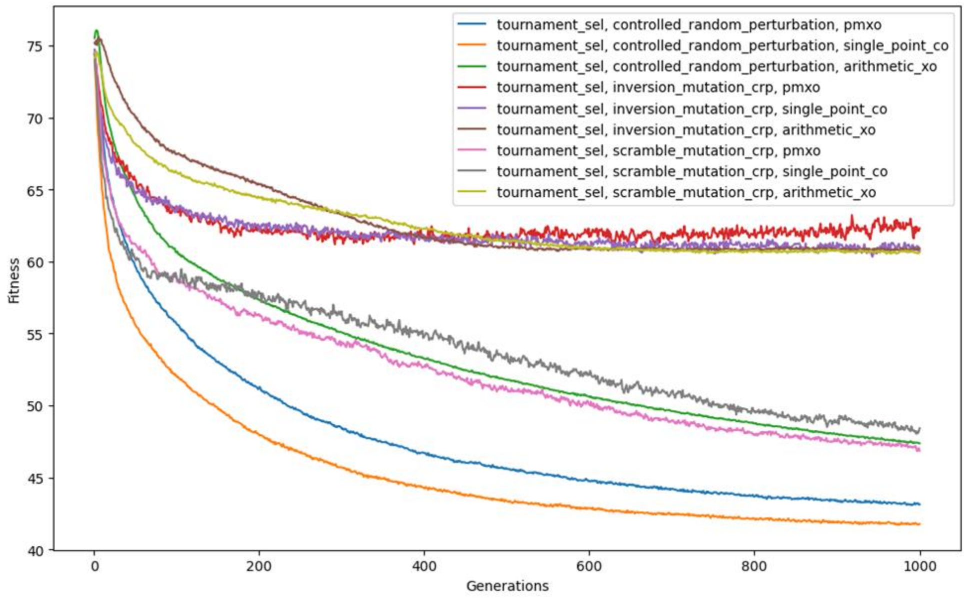


Annex Figure 1 - Results with CRP = 1

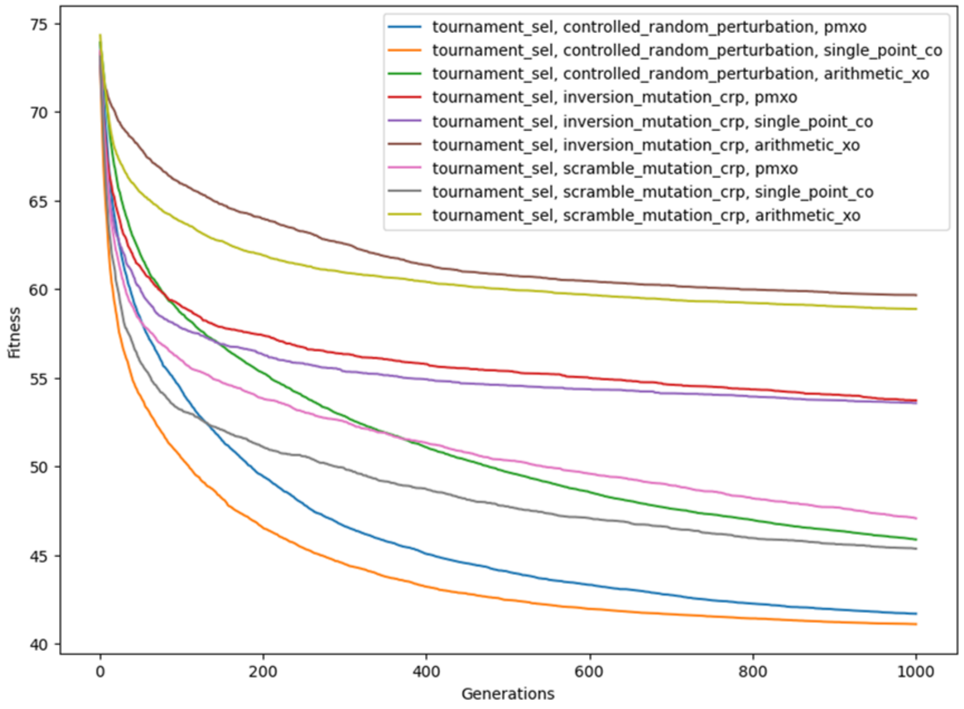


Annex Figure 2 - Results with CRP = 0.95

Annex Figures 3 and 4: Comparison of the algorithm results with elitism and without



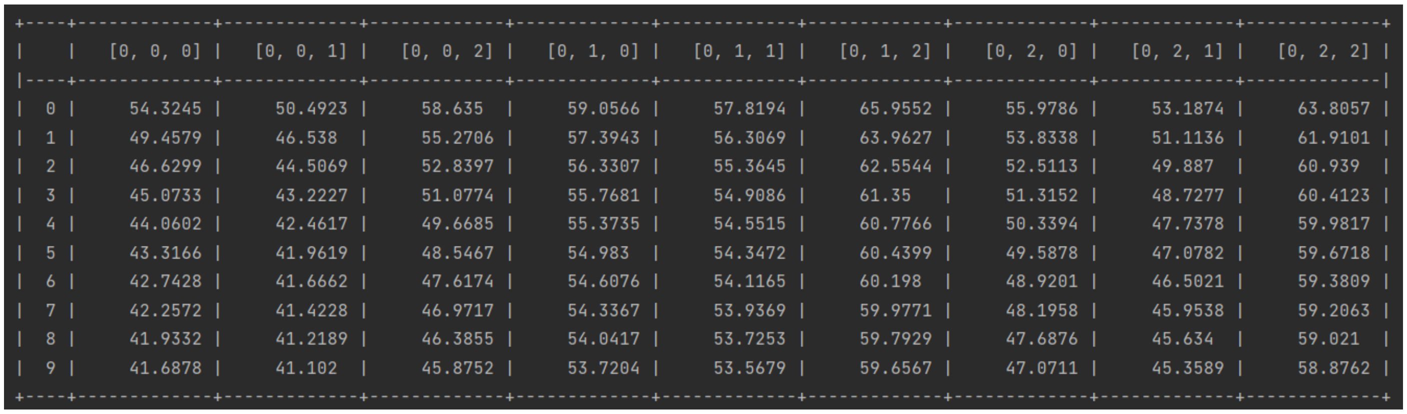
Annex Figure 3 - Results with Elitism = **False**



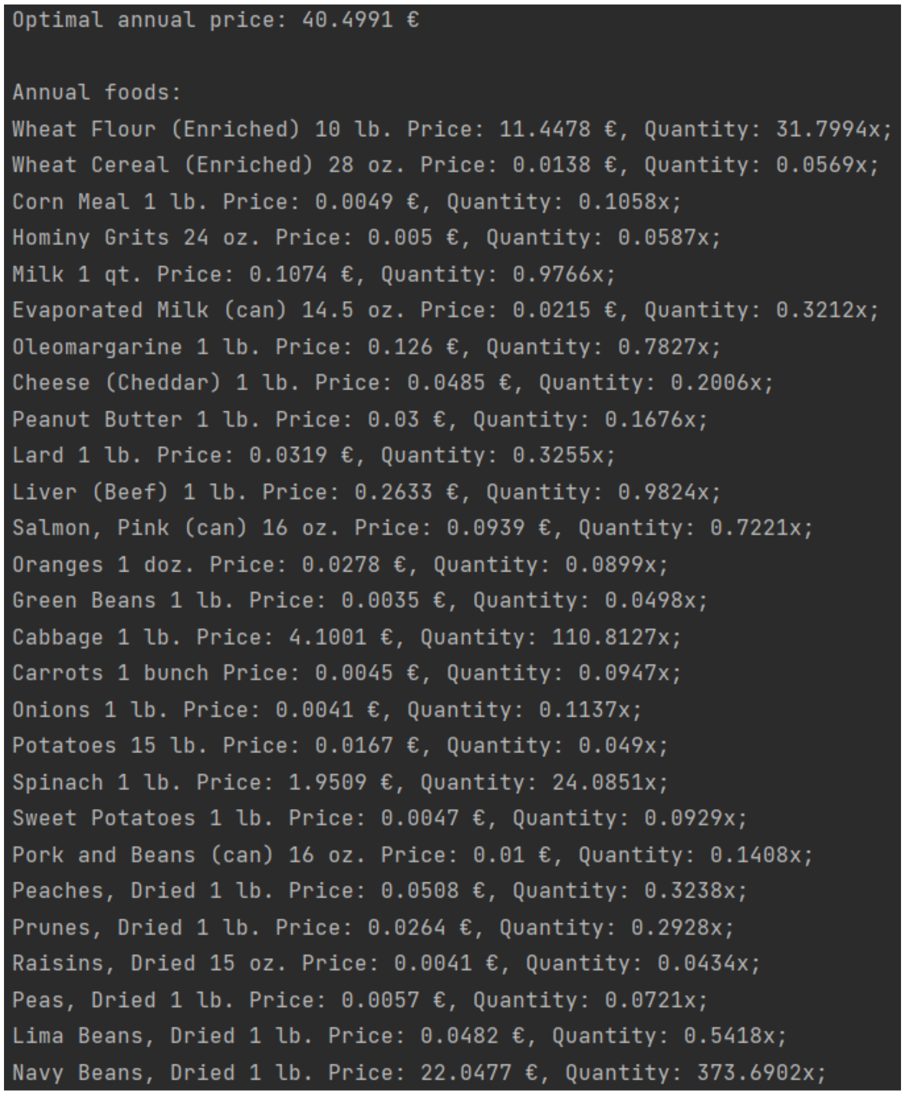
Annex Figure 4 - Results with Elitism = **True**

Annex Figure 5: Average Fitness / 100 generations for all solutions

First Index is selection method [0 = tournament_sel]
Second Index is mutation [0 = crp, 1 = inversion_mutation_crp, 2 = scramble_mutation_crp]
Third Index is crossover [0 = pmxo, 1 = single_point_xo, 2 = arithmetic_xo]

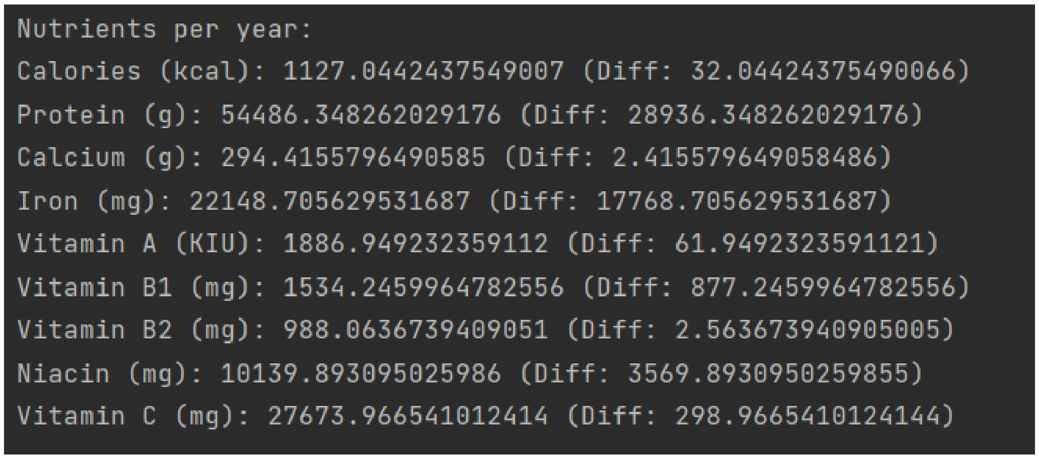

```
+----+------------+------------+------------+------------+------------+------------+------------+------------+------------+
|    |  [0, 0, 0] |  [0, 0, 1] |  [0, 0, 2] |  [0, 1, 0] |  [0, 1, 1] |  [0, 1, 2] |  [0, 2, 0] |  [0, 2, 1] |  [0, 2, 2] |
|----+------------+------------+------------+------------+------------+------------+------------+------------+------------|
|  0 |    54.3245 |    50.4923 |    58.635  |    59.0566 |    57.8194 |    65.9552 |    55.9786 |    53.1874 |    63.8057 |
|  1 |    49.4579 |    46.538  |    55.2706 |    57.3943 |    56.3069 |    63.9627 |    53.8338 |    51.1136 |    61.9101 |
|  2 |    46.6299 |    44.5069 |    52.8397 |    56.3307 |    55.3645 |    62.5544 |    52.5113 |    49.887  |    60.939  |
|  3 |    45.0733 |    43.2227 |    51.0774 |    55.7681 |    54.9086 |    61.35   |    51.3152 |    48.7277 |    60.4123 |
|  4 |    44.0602 |    42.4617 |    49.6685 |    55.3735 |    54.5515 |    60.7766 |    50.3394 |    47.7378 |    59.9817 |
|  5 |    43.3166 |    41.9619 |    48.5467 |    54.983  |    54.3472 |    60.4399 |    49.5878 |    47.0782 |    59.6718 |
|  6 |    42.7428 |    41.6662 |    47.6174 |    54.6076 |    54.1165 |    60.198  |    48.9201 |    46.5021 |    59.3809 |
|  7 |    42.2572 |    41.4228 |    46.9717 |    54.3367 |    53.9369 |    59.9771 |    48.1958 |    45.9538 |    59.2063 |
|  8 |    41.9332 |    41.2189 |    46.3855 |    54.0417 |    53.7253 |    59.7929 |    47.6876 |    45.634  |    59.021  |
|  9 |    41.6878 |    41.102  |    45.8752 |    53.7204 |    53.5679 |    59.6567 |    47.0711 |    45.3589 |    58.8762 |
+----+------------+------------+------------+------------+------------+------------+------------+------------+------------+
```

Annex Figure 6: List of foods and amounts for best solution (not average)


```
Optimal annual price: 40.4991 €

Annual foods:
Wheat Flour (Enriched) 10 lb. Price: 11.4478 €, Quantity: 31.7994x;
Wheat Cereal (Enriched) 28 oz. Price: 0.0138 €, Quantity: 0.0569x;
Corn Meal 1 lb. Price: 0.0049 €, Quantity: 0.1058x;
Hominy Grits 24 oz. Price: 0.005 €, Quantity: 0.0587x;
Milk 1 qt. Price: 0.1074 €, Quantity: 0.9766x;
Evaporated Milk (can) 14.5 oz. Price: 0.0215 €, Quantity: 0.3212x;
Oleomargarine 1 lb. Price: 0.126 €, Quantity: 0.7827x;
Cheese (Cheddar) 1 lb. Price: 0.0485 €, Quantity: 0.2006x;
Peanut Butter 1 lb. Price: 0.03 €, Quantity: 0.1676x;
Lard 1 lb. Price: 0.0319 €, Quantity: 0.3255x;
Liver (Beef) 1 lb. Price: 0.2633 €, Quantity: 0.9824x;
Salmon, Pink (can) 16 oz. Price: 0.0939 €, Quantity: 0.7221x;
Oranges 1 doz. Price: 0.0278 €, Quantity: 0.0899x;
Green Beans 1 lb. Price: 0.0035 €, Quantity: 0.0498x;
Cabbage 1 lb. Price: 4.1001 €, Quantity: 110.8127x;
Carrots 1 bunch Price: 0.0045 €, Quantity: 0.0947x;
Onions 1 lb. Price: 0.0041 €, Quantity: 0.1137x;
Potatoes 15 lb. Price: 0.0167 €, Quantity: 0.049x;
Spinach 1 lb. Price: 1.9509 €, Quantity: 24.0851x;
Sweet Potatoes 1 lb. Price: 0.0047 €, Quantity: 0.0929x;
Pork and Beans (can) 16 oz. Price: 0.01 €, Quantity: 0.1408x;
Peaches, Dried 1 lb. Price: 0.0508 €, Quantity: 0.3238x;
Prunes, Dried 1 lb. Price: 0.0264 €, Quantity: 0.2928x;
Raisins, Dried 15 oz. Price: 0.0041 €, Quantity: 0.0434x;
Peas, Dried 1 lb. Price: 0.0057 €, Quantity: 0.0721x;
Lima Beans, Dried 1 lb. Price: 0.0482 €, Quantity: 0.5418x;
Navy Beans, Dried 1 lb. Price: 22.0477 €, Quantity: 373.6902x;
```

Annex Figure 6: Nutrients achieved by best solution; *Diff* represents surplus



```
Nutrients per year:
Calories (kcal): 1127.0442437549007 (Diff: 32.04424375490066)
Protein (g): 54486.348262029176 (Diff: 28936.348262029176)
Calcium (g): 294.4155796490585 (Diff: 2.415579649058486)
Iron (mg): 22148.705629531687 (Diff: 17768.705629531687)
Vitamin A (KIU): 1886.949232359112 (Diff: 61.9492323591121)
Vitamin B1 (mg): 1534.2459964782556 (Diff: 877.2459964782556)
Vitamin B2 (mg): 988.0636739409051 (Diff: 2.563673940905005)
Niacin (mg): 10139.893095025986 (Diff: 3569.8930950259855)
Vitamin C (mg): 27673.966541012414 (Diff: 298.9665410124144)
```