Department of Computer Science

Ben Gurion University of the Negev

Evolutionary Algorithms

Winter 2023

Prof. Moshe Sipper

Find the Perfect Meal

Noa Leron

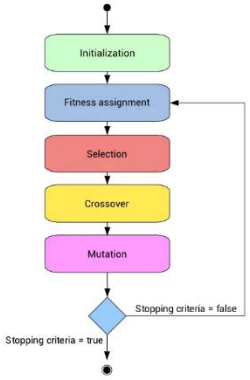
Yarden Kantor

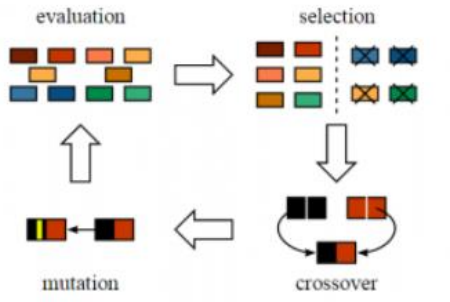
Table of Content:

Introduction:

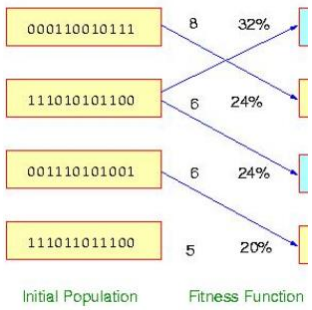
Evolutionary algorithms are a subset of artificial intelligence that are inspired by the process of natural evolution. They are used to find approximate solutions to optimization and search problems. Some popular examples of evolutionary algorithms include genetic algorithms, evolutionary strategies and particle swarm optimization. We will focus on genetic algorithms.

A genetic algorithm is a method based on the principles of natural selection and genetics, for solving search and optimization problems. It is a heuristic search method that imitates the process of natural selection in order to find the best solution for a problem. The algorithm starts with a population of solutions, and iteratively applies genetic operators (such as selection, crossover and mutation), to the population, creating a new generation of solutions. The solutions are evaluated based on a fitness function, and the most fit solutions are more likely to be selected, the selected solutions create the new generation. The process continues until a satisfactory solution is found or until a stopping criterion is met. Genetic algorithms are commonly used in a wide range of fields, such as machine learning, image processing and engineering design.

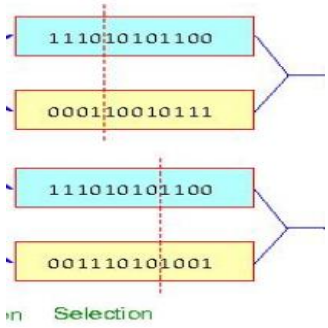




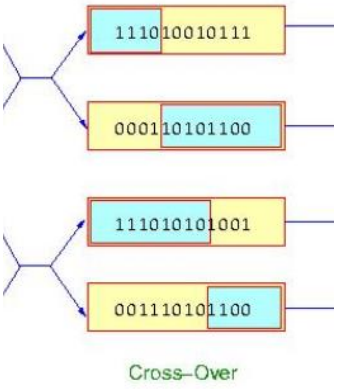
A fitness function is a function that assigns a fitness value to an individual solution. It measures the quality (fitness) of a solution in relation to the problem being solved. The fitness function is problem specific and is designed to evaluate how well a given solution performs against certain criteria. The fitness values are used as a guide for the genetic operators, to direct the search towards better solutions. The solutions with higher fitness values are more likely to be selected to create the next generation. The goal of the genetic algorithm is to find the solution with the highest fitness value, which is considered the optimal solution to the problem.



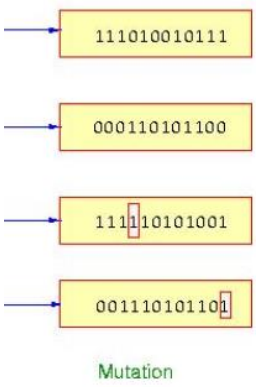
Selection is the process of choosing which solution will be allowed to “reproduce” and create the next generation. The solutions are typically chosen based on their fitness values, with the most fit solution being more likely to be selected. This helps to ensure the best solutions are passed on to the next generation.



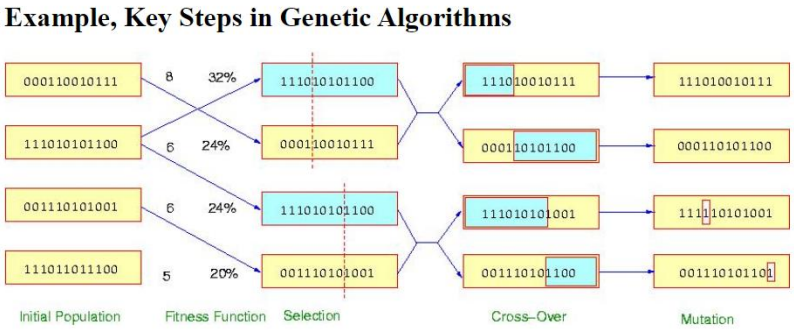
Crossover is the process of combining two or more solutions to create new solutions. It typically involves selecting two solutions from the current generation at random and then “crossing over” their genetic information to create a new solution. This can be thought of as a form of “mating” between solutions. The solutions created by crossover are intended to have a combination of the desirable characteristics of the parent solution.



Mutation is the process of creating small, random changes to a solution. It is used to introduce new genetic information into the population for the sake of maintaining genetic diversity. Without mutation, the population would become similar, and the search would become stuck in a local optimum.



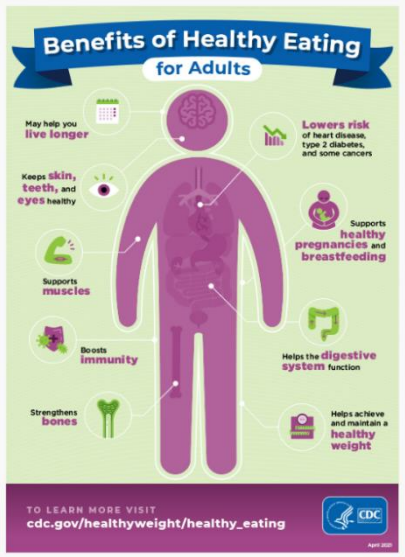
These genetic operators work together to explore the solution space and to find the optimal solution to the problem. Each generation of solutions is better than the previous one (Although some generations can be excluded), and the genetic algorithm continues until a stopping criterion is met, such as reaching a maximum number of generations or finding an acceptable solution.

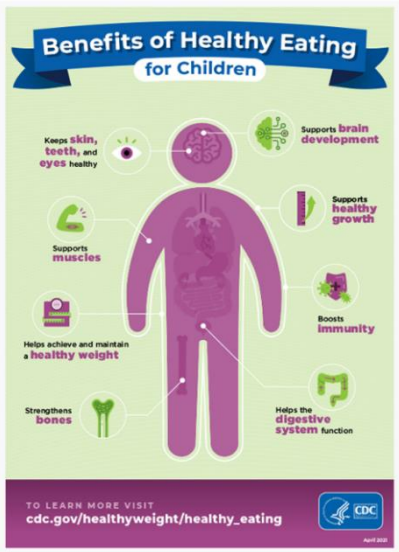


Find Perfect meal problem:

The "World Health Organization" describes the reasons for maintaining a healthy diet: *"It protects you against many chronic noncommunicable diseases, such as heart disease, diabetes and cancer. Eating a variety of foods and consuming less salt, sugars and saturated and industrially-produced trans-fats, are essential for healthy diet."* A healthy diet comprises a combination of different foods, such as staples, starchy tubers and roots, legumes, fruit and vegetables, and foods from animal source.

Benefits of healthy eating include living longer, keeping skin, teeth and eyes healthy, muscle support, immunity boost, strengthening bones, lowering the risk of heart disease, of type two diabetes and of some cancers, supporting healthy pregnancies and breastfeeding, helping the digestive system, achieving and maintaining a healthy weight.





From a nutritional standpoint, a perfect meal should provide a balance of nutrients such as carbohydrates, proteins, and fats. It should also be within one's daily caloric needs. From a taste perspective, a perfect meal should have a balance of flavors and textures, and be maid with fresh and high quality ingredients. It should be visually appealing and presented in an attractive manner.

One way to find the perfect meal is to experiment with different cuisines and ingredients, and to try new recipes. It can be grueling, exhausting and time consuming, many people can't or don't want to go through a long process, and just want a quick solution for their problem.

The formulation of the problem: given an enormous number of various options for assembling a meal, with a variety of nutritional values, scattered across the internet, How can one find a meal suitable for their nutritional needs?

Solution:

We're providing a web application that uses an evolutionary algorithm to find the perfect meal.

The user enters the importance percentage of each nutritional value (fats, protein and carbohydrates) and the maximum number of calories the user want's for the meal. The website presents a list of foods that combine a meal under the requested calories, with the most fitting nutritional values. The foods shown by the website were chosen using an evolutionary algorithm consisting of random choices, among others, therefore, if the user want's another option they can repeat the previous process (re-enter the properties for their perfect meal, and the algorithm will print different components to the screen.

The database of the various foods and ingredients, and their nutritional values, was taken from the *Food and Agriculture Organization of the United nations*. This database contains more than a thousand different foods an ingredients with their nutritional values, including Fat, Carbohydrates, Protein, Fiber, Alcohol, Ash, Calcium, Iron, Magnesium, Phosphorus, Potassium, Sodium, Zinc, Copper, Vitamin A, Retinol, Beta-carotene, Alfa-carotene, Beta-cryptoxanthin, Vitamin D, Vitamin E, Alfa-tocopherol, Beta-tocopherol, Gamma-tocopherol, Delta-tocopherol, Thiamine (Vitamin B1), Riboflavin (Vitamin B2), Niacin (Vitamin B3), Tryptophan, Vitamin B6, Folic acid, Folate, Vitamin B12, Vitamin C, Cholesterol, Fatty acids, Linoleic acids, Alfa-linoleic acid, Phytate, Inositol triphosphate, Inositol tetraphosphate, Inositol pentaphosphate and Inositol hexaphosphate. We chose to focus on the most known nutritional values- fats, carbohydrates, protein, and of course calories since these are the categories found to be most relevant to the majority of the population. However, the application and the evolutionary algorithm can be easily modified to include additional categories of nutritional values. The change in the algorithm will be reflected in the fitness function.

The fitness function we wrote, multiplies each category of numerical value (of a certain food) by the percentage entered by the user, then adds the product to the sum of all the other categories' products, then adds the sums of each food. If the sum of all the foods' calories is not larger than the maximum calories value entered by the user, then the fitness value of this solution is the sum of the sums mentioned above, else, the fitness value of that solution is minus infinity, where a higher fitness value is better. Adding other categories of nutritional values to the equation will be visible in adding more products to the sums.

The GABitStringVectorCreator function takes a generation creator function as one of it's arguments, the generation creator function returns a binary digit (0 or 1) in random, thus determines the value of a specific component of the bit vector. We will elaborate farther about the GABitStringVectorCreator in the next section. The default generation creator function returns the binary value 1 with a probability of 0.5. this probability was too high for our implementation of the genetic algorithm since we have a database of size 1028 and a meal under 300- 2000 calories (which is the range of the most common calory limit) consists of 5 – 20 foods. For our implementation the most suitable distribution is a discrete Bernoulli distribution $x \sim B[n,p]$ with a probability function of $\mathbb{P}(x=m)=\binom{n}{m}p^m(1-p)^{n-m}$. In our case, $n=1028, i$n order to find p , we must find the probability of getting the binary value 1 for a specific component, which have a uniform distribution $y \sim U[1,1600]$ with a probability of $\mathbb{P}(y=k)=\frac{1}{1600-1+1}$. Thence, the probability of the value 1 in a certain component is the same as the probability of k%1600=0 that is the probability of k being 1600 which is $p=\mathbb{P}(y=1600)=\frac{1}{1600-1+1}=\frac{1}{1600}$. Now the probability of getting a vector with m components with the value 1 is $\mathbb{P}(x=m)=\binom{n}{m}p^m(1-p)^{n-m}=\binom{1028}{m}(\frac{1}{1600})^m(\frac{1599}{1600})^{1028-m}$. To achieve the probability $p=\frac{1}{1600}$ we used the random.randint function with a range of 1–1600 to generate a number k within this range, then we return the binary value 0 if k%1600 isn't 0, and the binary value 1, otherwise. The Bernoulli distribution is achieved because we apply our generation creator function 1028 times, one for each vector component.

Software Overview:

In our implementation, an individual solution is represented as a bit vector of length 1028, each bit is of a binary value 1 if and only if the food of the matching component was chosen, and a binary value of 0, otherwise. Hence we chose the mutation operator to be a bit flip mutation and the crossover operator to be a binary k points crossover. The fitness function adds the calories of each chosen food in a solution, and if the sum exceeds the calories limit, the function returns a fitness value of minus infinity, because we didn't want the algorithm to choose a solution that doesn't meet the requirement of limiting the amount of calories in the meal. So we keep the promise to return a meal under the calories limit.

We chose to use the following functions in our implementation of the evolutionary algorithm, since they fit our representation of an individual solution.

SimpleEvolurion is a class from the *simplle_evalustion* module of the *EC-kity* package. The class is used to perform a simple form of evolution which is a metaheuristic optimization algorithm inspired by the process of natural evolution. The *SimpleEvolution* class provides a basic implementation of an evolutionary algorithm. It includes methods for creating an individual population, performing selection, crossover and mutation, and updating the population. It provides a method of evolving the population over multiple generations, as well. It can be costumed by providing the function to optimize, population size, mutation and crossover rate, selection method and other parameters. We chose a population size of 70, for instance, because it seemed to be the best size in our empirical results. We will expand on these functions and the other measures later.

SimpleBreeder is a class from the *simple_breeder* module of the *EC-kity* package. The class is used to perform the breeding process in an evolutionary algorithm, which is responsible for creating new individuals in the population. The *SimpleBreeder* class includes methods for performing selection, crossover and mutation on the individuals in the population, and updating the population. It can be modified by providing the mutation and crossover rate, selection method, and other parameters.

GABitStringVectorCreation is a class from the *bit_string_vector_creator* of the *EC-kity* package. The class is used to create new individuals for a population in a genetic algorithm, specifically for bit string representation. The *GABitStringVectorCreation* class provides a method for generating new indeviduals in the population in the form of bit strings. The class has a method that randomly generates bit strings of a specified length, this class allows to set the length of the bit string, which can be used as an attribute of the individual. We chose the vector length to be 1028 since it's the size of our database. This class uses a random generation creator function, we provided a costume generation creator function that suits the Bernoulli distribution as mentioned before.

VectorKPointsCrossover is a class from the *vector_k_points_crossover* of the *EC-kity* package. The class is used to perform a specific type of crossover called "k-points crossover" on vectors. In k-points crossover, a specified number k of random "cut points" are chosen in the parents' vectors, and the bits of the vectors are then exchanged between the parent at these points. This operation is used to create new individuals by combining the genetic information from the parents. We chose the crossover probability to be 0.5 and to have k=2 crossover point, we tried to apply the algorithm with different probability values and found that 0.5 gave the most stable results, different k values didn't have an impact on the results, because most bits had a zero value due to the generation creator function.

BitStringVectorFlipMutation is a class from the *vector_random_mutation* module of the *EC-kity* package. The class is used to perform a specific type of mutation operation called "flip mutation" on

bit strings, which are sequences of binary digits. In flip mutation, a random bit in the bit string is selected and it's value is flipped (i.e. 0 becomes 1 or vice versa). This operation is used to introduce variation to the population of bit strings. We chose the probability of flipping some bit to be 0.05, all of the empirical results for the different probability values was similar, because most bits had a zero value due to the generation creator function.

ToutnamentSelection is a class from the *tournament_selection* module of the *EC-kity* package. The class is used to perform a specific type of selection method called "Tournament selection" on individuals in a population. In tournament selection, a specified number of individuals are randomly chosen from the population and compete against each other. The individual with the highest fitness is then selected as the parent for the next generation. This operation is used to select the best individual from the population to be used for breeding. The *ToutnamentSelection* class can be customized with different parameters such as the number of individuals participating in the tournament and the tournament size. We chose to have a tournament size of 4 since it was the largest size we could use and still have a reasonable run time of the algorithm.

BestAveregeWorstStatistics is a class from the *best_averege_worst_statistics* module of the *EC-kity* package. The class is used to calculate various statistics such as the best, average an worst fitness of the population in a given generation of an evolutionary algorithm. This class provides a method to calculate the best, average an worst fitness of the population. These values can be used to track the progress of an evolutionary algorithm and to evaluate the performance. When experimenting and running the algorithm we observed that the algorithm gets good results after a relatively small number of generations. Hence, for a better run time, we chose to have 5 generations, we will elaborate about this choice later on. We did notice better results were achieved over more generation, thus decided to provide an option of choosing the number of generations.

Subpopulation is a class from the *subpopulation* module of the *EC-kity* package. The class is used to create and manage a subpopulation, which is a subset of the main population in the evolutionary algorithm. This class allows to create a new subpopulation from a main population and to manage it, for example it allows to update the subpopulation and get the individuals. The *subpopulation* class can be useful in different scenarios such as in Multi-population genetic algorithms, or to separate a population into different subpopulations to perform specific operations on them. Since we chose to use the *SimpleEvolurion* class, we only had one subpopulation, but there wasn't any good reason to use more than one subpopulation in our implementation, so we didn't have some of the scenarios mentioned above.

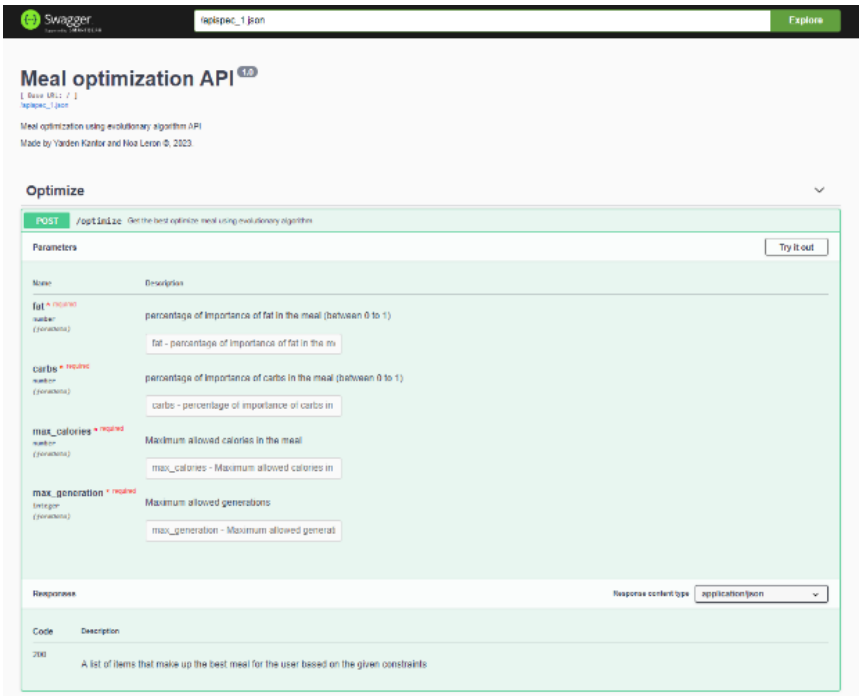
After writing the evolutionary algorithm, running and testing it, we had to choose how to use our evolutionary algorithm in a web application. We decided the best approach for our purposes was complete separation of backend and frontend, since the backend was written in python, due to the requirement use the *EC-kity* package. We rapped the python code with *Flask* for the sake of using *Python* and *EC-kity*, and with a modern *APIDOC*. The frontend part was written using REACT, which is a modern platform of writing an applicative infrastructure with optimal performance, it is newer and has better performance the *Angular,* for comparison. We than uploaded the frontend part to *Netlify*, a free web hosting platform, and uploaded the backend part to *Render,* a cloud application hosting platform. The reason for this separation is minimizing the run time of the algorithm even though our application runs on a free environment. The separation was not enough for maintaining a reactive user interface, thus, for the purpose of providing a reasonable user experience, we had to limit the number of generations. Despite this limitation, we wanted to demonstrate a better performance of the algorithm when running on an enabling environment, so we set up a private server on a simple *Raspberry Pi* infrastructure. We used a *Raspberry Pi 4 Module B, 8GB LPDDR4-3200 SDRAM with*

Broadcom BCM2711 processor (ARM64) with Raspberry pi OS of 64-bit, kernel version 5.15, Debian version 11 bullseye, which runs an optimal *Linux* environment. We set up the code in a process in the *Linux* operating system of the *Raspberry Pi* by adding two simple scripts to the service manager of the operating system, then, after we configured the infrastructure (since it's a private home infrastructure), our system is finally accessible through these two addresses:

http://eveolutionary-algo.serveirc.com:9000/

http://eveolutionary-algo.serveirc.com:8000/apidocs/

The second one (with *Raspberry Pi* infrastructure) has better performance, so we added an option of choosing the number of generations for running the evolutionary algorithm.



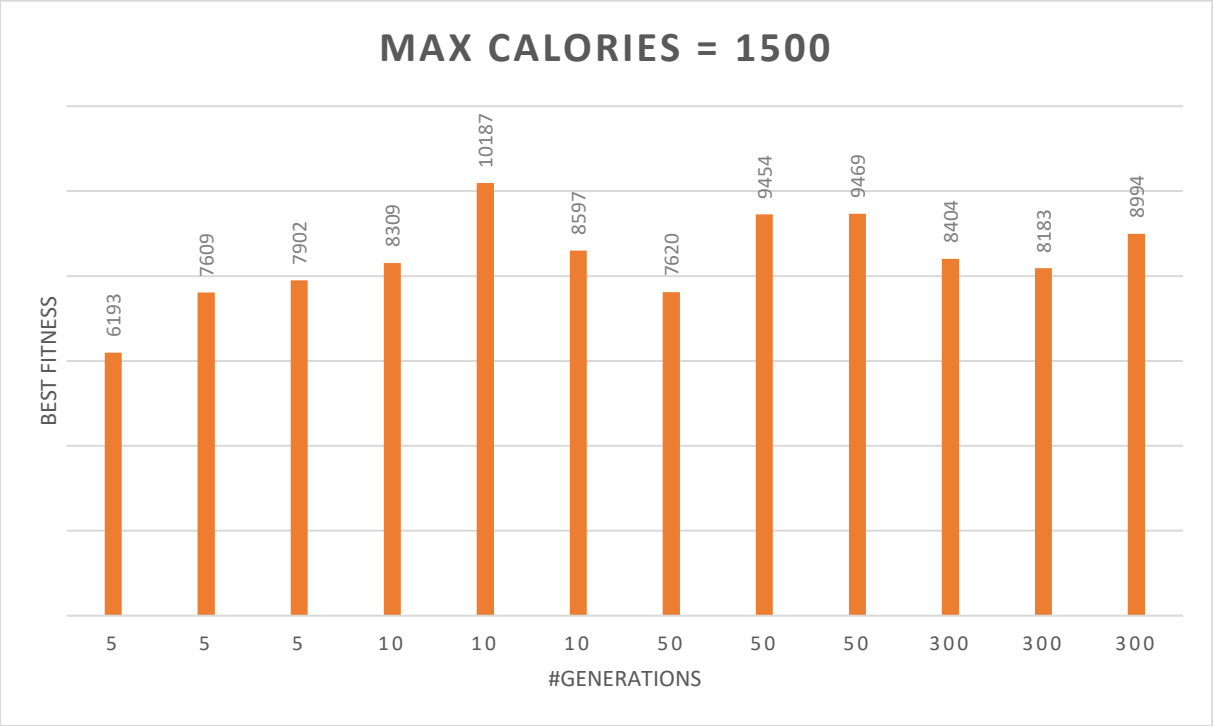
Experiments and Results:

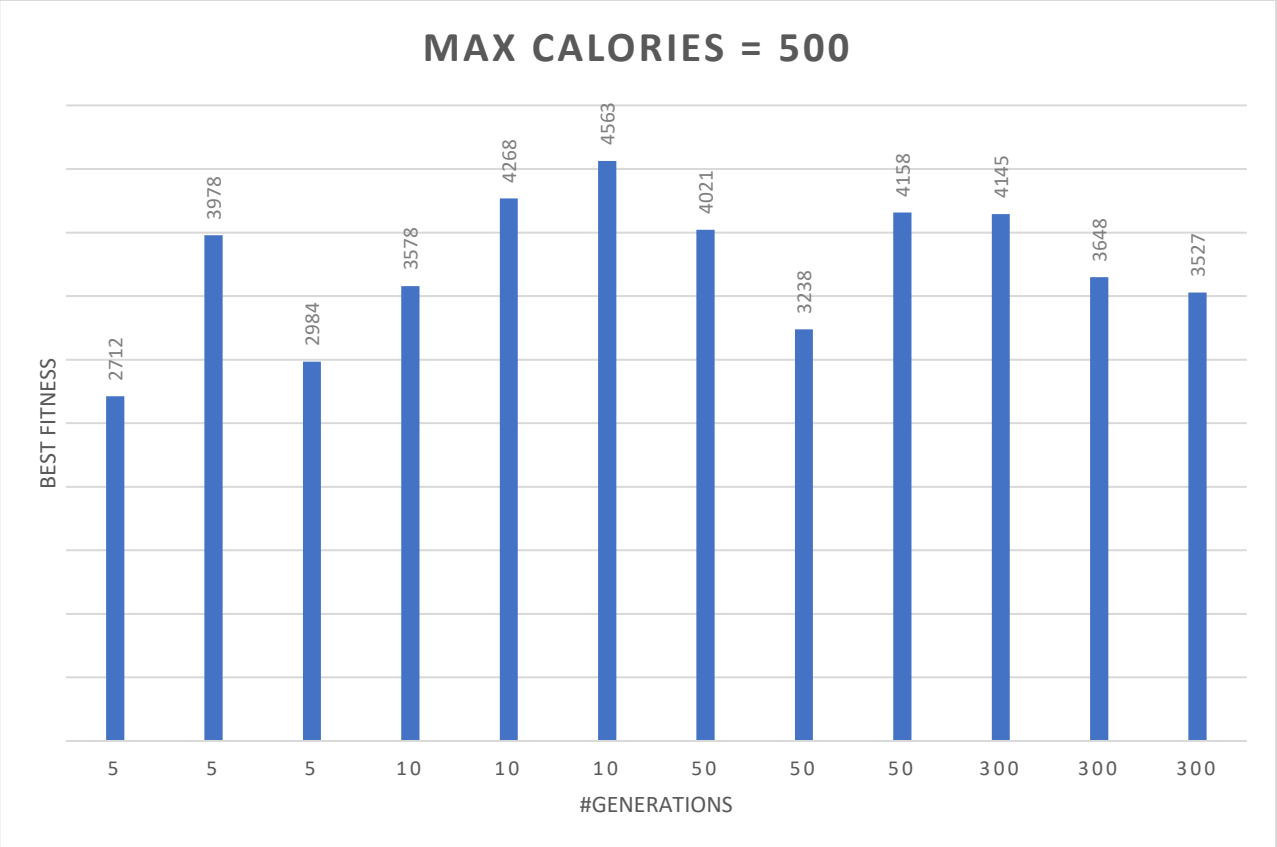
We ran the algorithm with different number of generations, with different calory limit, we chose to run it with the same fat, carbs and protein values (fat = 10, carbs = 10, protein = 50) since these variables affect the fitness value, yet their effect is irrelevant when determining the minimum number of generations that will provide optimal results.

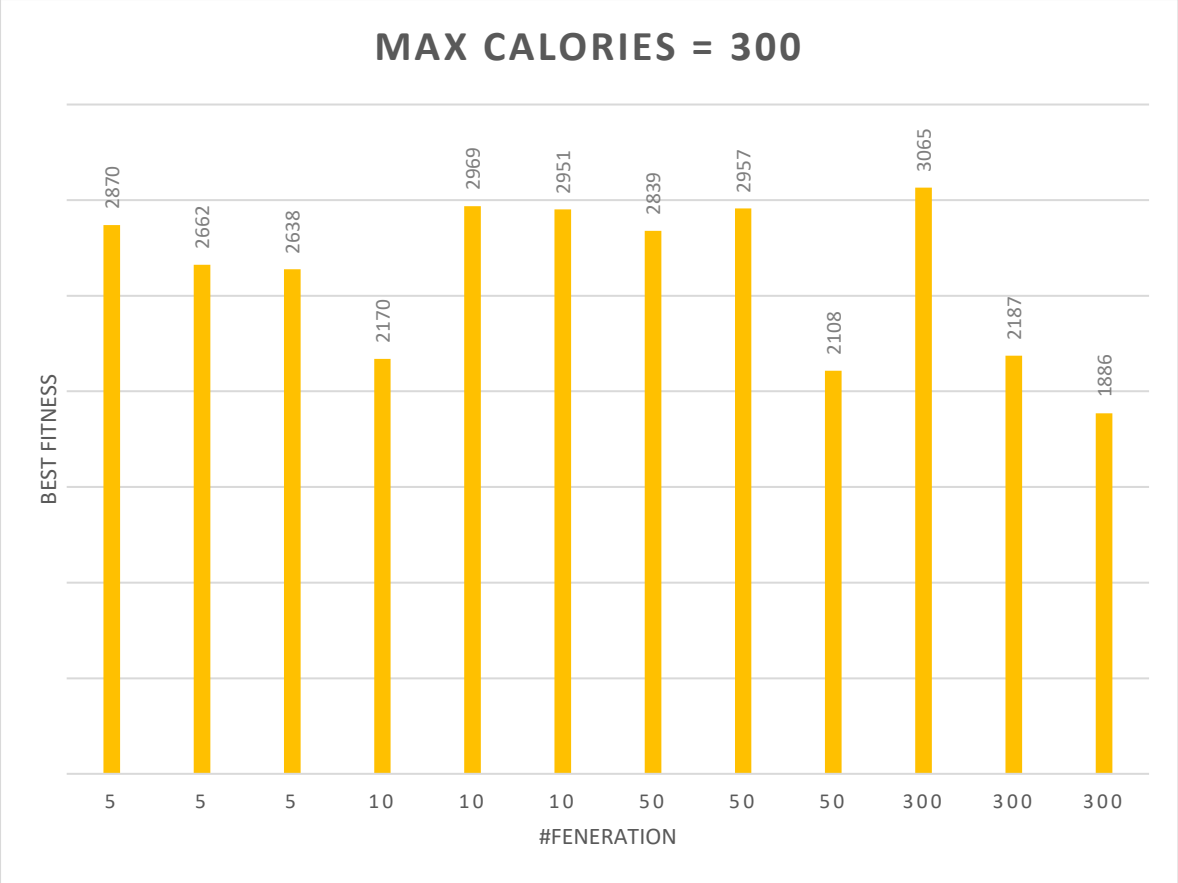
The following table demonstrates the best, worst and average fitness values obtained when running the evolutionary algorithm, over various number of generations with different calory limits:

Iteration	#Generations	Best Fitness	Worst Fitness	Average Fitness	Fat	Carbs	Protein	Max Calories
1	5	6193	-∞	-∞	10	10	50	1500
2	5	7609	2541	4597.628	10	10	50	1500
3	5	7902	-∞	-∞	10	10	50	1500
4	10	8309	-∞	-∞	10	10	50	1500
5	10	10187	-∞	-∞	10	10	50	1500
6	10	8597	-∞	-∞	10	10	50	1500
7	50	7620	-∞	-∞	10	10	50	1500
8	50	9454	-∞	-∞	10	10	50	1500
9	50	9469	-∞	-∞	10	10	50	1500
10	300	8404	-∞	-∞	10	10	50	1500
11	300	8183	-∞	-∞	10	10	50	1500
12	300	8994	-∞	-∞	10	10	50	1500
13	5	2712	-∞	-∞	10	10	50	500
14	5	3978	-∞	-∞	10	10	50	500
15	5	2984	-∞	-∞	10	10	50	500
16	10	3578	-∞	-∞	10	10	50	500
17	10	4268	-∞	-∞	10	10	50	500
18	10	4563	4554	4554.1258	10	10	50	500
19	50	4021	-∞	-∞	10	10	50	500
20	50	3238	-∞	-∞	10	10	50	500
21	50	4158	-∞	-∞	10	10	50	500
22	300	4145	-∞	-∞	10	10	50	500
23	300	3648	-∞	-∞	10	10	50	500
24	300	3527	-∞	-∞	10	10	50	500
25	5	2870	-∞	-∞	10	10	50	300
26	5	2662	-∞	-∞	10	10	50	300
27	5	2638	-∞	-∞	10	10	50	300
28	10	2170	-∞	-∞	10	10	50	300
29	10	2969	2969	2969	10	10	50	300
30	10	2951	-∞	-∞	10	10	50	300
31	50	2839	-∞	-∞	10	10	50	300
32	50	2957	-∞	-∞	10	10	50	300
33	50	2108	-∞	-∞	10	10	50	300
34	300	3065	-∞	-∞	10	10	50	300
35	300	2187	-∞	-∞	10	10	50	300
36	300	1886	-∞	-∞	10	10	50	300

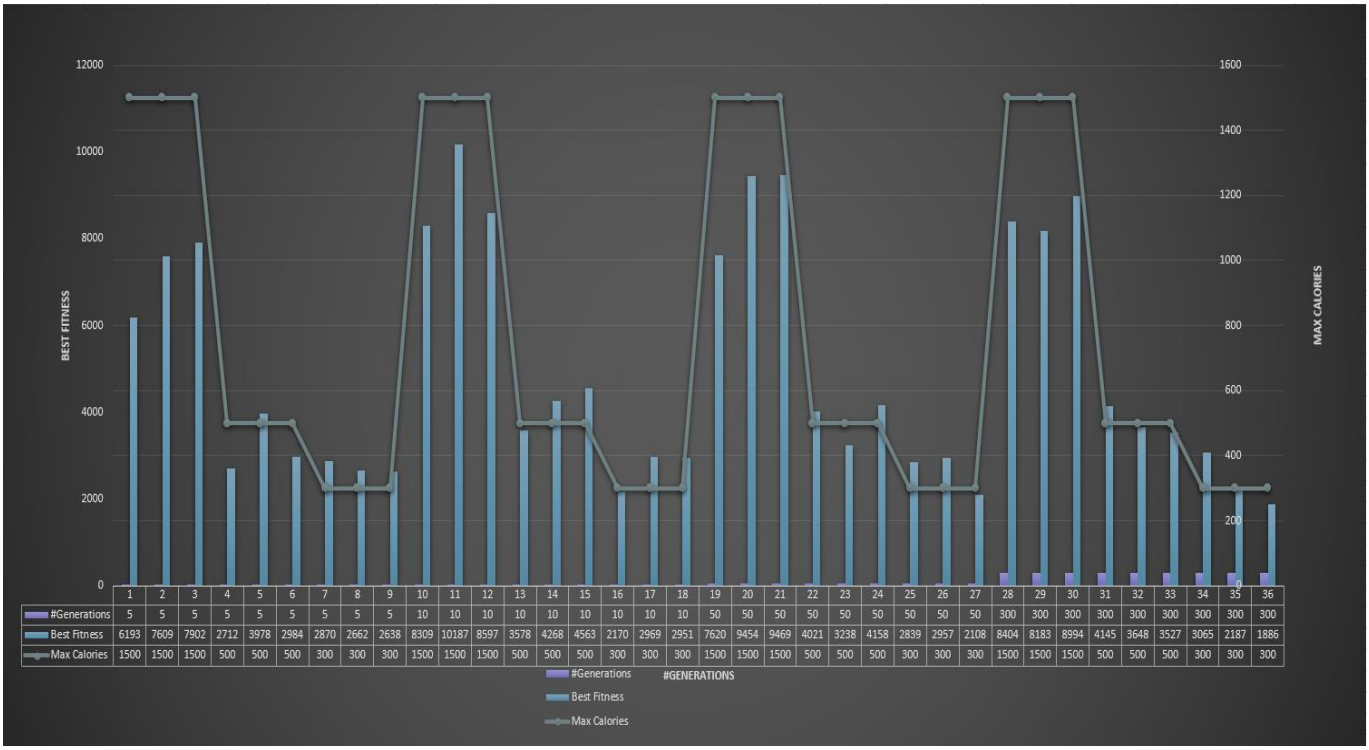
The following charts represent the fitness values as a function of the number of generations for running the evolutionary algorithm to find a meal under 1500, 500, 300 calories respectively:





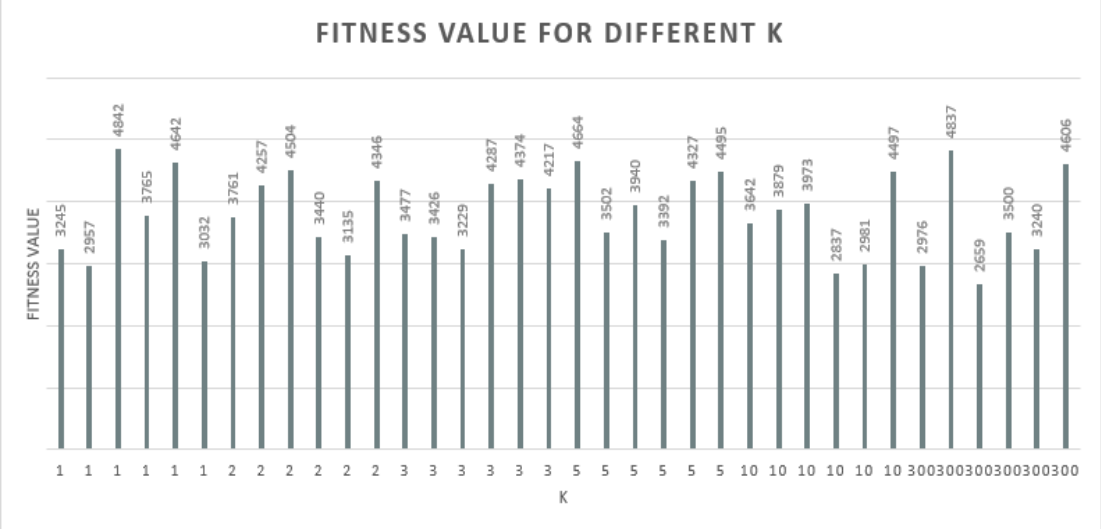


The following chart represents the fitness values as a function of the number of generations for running the evolutionary algorithm with a secondary axis of the calory limit value:



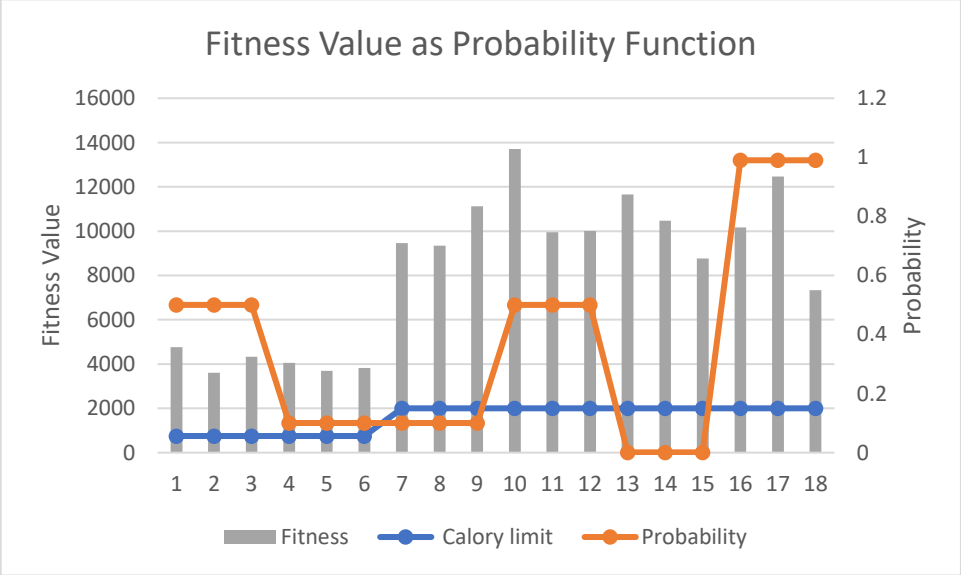
The following chart and table show the different fitness values as a function of the k variable of the *VectorKPointCrossover* function:

Iteration	# Generations	K	Best Fitness	Calory Lir
1	10	1	3245	500
2	10	1	2957	500
3	10	1	4842	500
4	50	1	3765	500
5	50	1	4642	500
6	50	1	3032	500
7	10	2	3761	500
8	10	2	4257	500
9	10	2	4504	500
10	50	2	3440	500
11	50	2	3135	500
12	50	2	4346	500
13	10	3	3477	500
14	10	3	3426	500
15	10	3	3229	500
16	50	3	4287	500
17	50	3	4374	500
18	50	3	4217	500
19	10	5	4664	500
20	10	5	3502	500
21	10	5	3940	500
22	50	5	3392	500
23	50	5	4327	500
24	50	5	4495	500
25	10	10	3642	500
26	10	10	3879	500
27	10	10	3973	500
28	50	10	2837	500
29	50	10	2981	500
30	50	10	4497	500
31	10	300	2976	500
32	10	300	4837	500
33	10	300	2659	500
34	50	300	3500	500
35	50	300	3240	500
36	50	300	4606	500



The following chart and table show the different fitness values as a function of the probability variable of the *VectorKPointCrossover* function:

Iteration	# Generation	Calory limit	Probability	Fitness
1	50	750	0.5	4756
2	50	750	0.5	3604
3	50	750	0.5	4326
4	50	750	0.1	4059
5	50	750	0.1	3691
6	50	750	0.1	3817
7	50	2000	0.1	9454
8	50	2000	0.1	9337
9	50	2000	0.1	11124
10	50	2000	0.5	13708
11	50	2000	0.5	9945
12	50	2000	0.5	10006
13	50	2000	0.001	11655
14	50	2000	0.001	10473
15	50	2000	0.001	8766
16	50	2000	0.99	10170
17	50	2000	0.99	12459
18	50	2000	0.99	7339



Conclusions:

The charts above represent the different fitness values achieved by applying the evolutionary algorithm for sundry values of the max_generation variable. They show that no significant changes in the results were achieved by applying the algorithm on larger values of the max_generation variable. That is the reason we chose to have one implementation of the web application with a low value of 5 generations. A slight increase of the fitness value can be observed with higher max_generation values, therefore, the user of our second implementation of the website is given the option to choose the max_generation value.

As shown in the above "Fitness Value for Different K" chart and table, the value K of the *VectorKPointCrossover* function have a little to no affect on the fitness value, we chose K to be 2 for simplification.

The above "Fitness Value as Probability Function" chart and table shows different values of the

the probability variable of the *VectorKPointCrossover* function gave a variety of results. meaning, very high and very low probability values, gave diversified fitness values for different applications of the algorithm, for the same probability value. But the probability value of 0.5 gave similar results in each of the applications of the algorithm, and that is why we chose 0.5 to be the value of the probability variable.

Bibliography:

- Worlds Health Organization: https://www.worldpediatricproject.org/?gclid=Cj0KCQiA_bieBhDSARIsADU4zLf_JVFrbpKb1lorZ5huz0dKgqUwRpFVhGEbln3pG0aLEz3d7kBBK0waAk6qEALw_wcB
- Center for Disease Control and Prevention: https://www.cdc.gov/nutrition/resources-publications/benefits-of-healthy-eating.html
- React Documentation: https://reactjs.org/
- Netlify Documentation: https://www.netlify.com/?utm_source=google&utm_medium=paid_search&utm_campaign=12755510784&adgroup=118788138897&utm_term=netlify&utm_content=kwd-309804753741&creative=514583565825&device=c&matchtype=b&location=9058761&gclid=Cj0KCQiA_bieBhDSARIsADU4zLfTubDVuvxYPt_rgLu7xXmcqQUSpgdVqq5jgYaEqCHbxk5kdGqp1FsaAvSCEALw_wcB
- Render: Cloud Application Hosting for Developers: https://render.com/
- APIDOC: https://apidocjs.com/
- ChatGPT: https://chat.openai.com/chat