Computational Biology – Decoding a Monoalphabetic Cipher

Dina Englard, 314632811- Bioinformatics

Aviel Zecharia, 207456641 - Computer science

**תרגיל משולב מדמח וביולוגיה חישובית**

*How to run:*

* Link to github which includes all the code:
  + <https://github.com/DinaEnglard/CompBio_Targil2/>
* Download solver.exe.
* Make sure dict.txt and enc.txt are in the same file.
* Run "solver.exe regular" for the regular algorithm.
* Run "solver.exe lamarck" for lamarckian algorithm.
* Run "solver.exe darwin" for darwin algorithm.

In the following paragraphs we will shortly describe our genetic algorithm and its functions:

We get an encoded text file as input. Our output is the solution: a permutation of the alphabet resulting in highest fitness score and the decoded text.

The fitness score is calculated as following:

*Fitness Function:*

Calculate the rating of each trigram (triplet of letters) based on a rating chart found online *here (https://github.com/David-Hinschberger/Monoalphabetic-Cipher-Decoder/blob/master/ratings.py).*

Score is the sum of all ratings for each triplet of letters in the text, after being translated according to the permutation in question.

*Algorithm:*

We start by creating an initial population made up of random permutations of the alphabet saved in form of a list.

We chose our population size to be 200 to balance high accuracy and low runtime.

We rate each permutation (perm) and save the perms and their scores in a dictionary.

We choose the top 10 performers and create the next generation as following:

* Add top 10 performers (elite population) to the new generation.
* Then, for the rest of the population(population\_size-elite\_population\_size):
  + take 2 perms (parents) randomly from elite population.
  + Crossover them: randomly select 2 indexes and replace the letters in parent 1 between that range of indexes with parent 2 in those indexes.
  + Repair the perm so it is valid alphabet perm: check which letters are missing, shuffle them, and then place them instead of letters that appear twice.
  + Mutate: each letter has a chance to be replaced with a different random letter. Our mutation rate is 0.1 (chosen by process of trial and error.)
  + return the child

Once we have the population of the right size, (made up of the 10 elite from the last run, and their children created in the process explained above) we rate the permutations in the population and sort them by descending order. Again, we choose the top 10 performers, using them to create the next generation in the same manner.

Our algorithm continues to run unless one of the following occurs:

Either we have exceeded 2000 generations, or our score exceeded 8.

(From the different tests we ran, score 6 usually resulted in a correct translation. Obviously, this is a hyperparameter which can be modified. )

*Lamarck Algorithm:*

For the Lamarck algorithm, we modified our regular algorithm as following:

After creating the initial random population, we mutated each perm by switching two random letters, checking if the score improved, and keeping the higher of the two. We repeated this process twice. (SWAPS =2 )

Next, each time we created a new child to add to the population, we first tried mutating it in the same way, again, saving the perm with the better result.

This way, we ran 2 local optimizations on each perm and sent the optimized perms to the next gen.

*Darwin Algorithm:*

Here, we used the same optimization technique as in the Lamarck algorithm, however, we used the optimization to EVALUATE the key, and sent the ORIGINAL keys (which gave the highest results in optimiziation. ) We did this by taking the initial random population, and changing the score of each key to the best score given in optimization, yet leaving the key as it was before optimization. (Sometimes, it remained the same.) Then, we chose the top 10 keys to pass on to the next generation. Once they were passed on to the elite population, we re-evaluated them and ordered them by their current scores. We repeated this for every new generation. We set the scores of the perms based on the best score received after optimization, yet kept the key as it was before optimization, passed the 10 best performers on to the next gen, and then re-evaluated them and ordered them in order of their current correct scores.

*Early convergence problem:*

We found that on shorter texts, we had a problem of early convergence. We decided to check when the best score remained the same for 50 generations. When this happened, we tried many solutions, including:

* randomly creating a new population and starting from the beginning, every time
* adding in random permutations in addition to the 10 perms (that didn't work since the elite ones almost always remained at the top.)

In the end, we decided to break the loop, stopping if we exceeded 50 generations with the same score, and returned the best result.

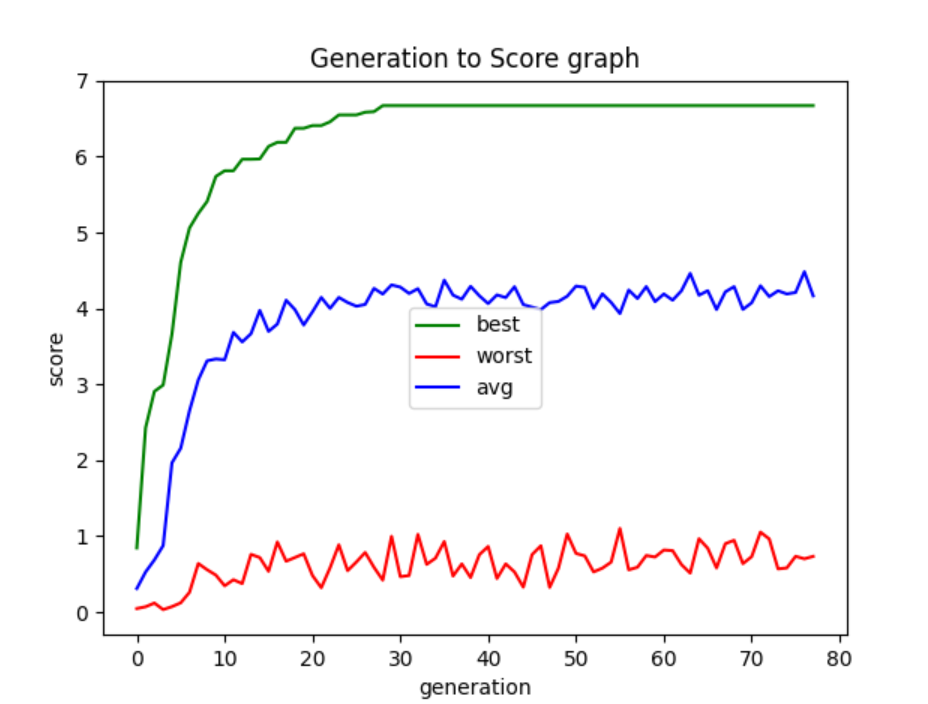
***Algorithms Analysis:***

To assess our algorithm, we checked how many words from plain.txt are valid words according to dict.txt. We validated ourselves by checking plain.txt with the eye. We saw that the best result is 90%, when all the words are valid, therefore meaning that some of the words aren’t in the dictionary provided.

Here are examples of all 3 algorithms run once, and a graph to plot the score as function of generation. Afterwards, we will show graphs of each algorithm run multiple times, and their progress over multiple executions.   
  
**REGULAR**

77 generations until convergence. Reached best score after 27 generations.

A black screen with white text

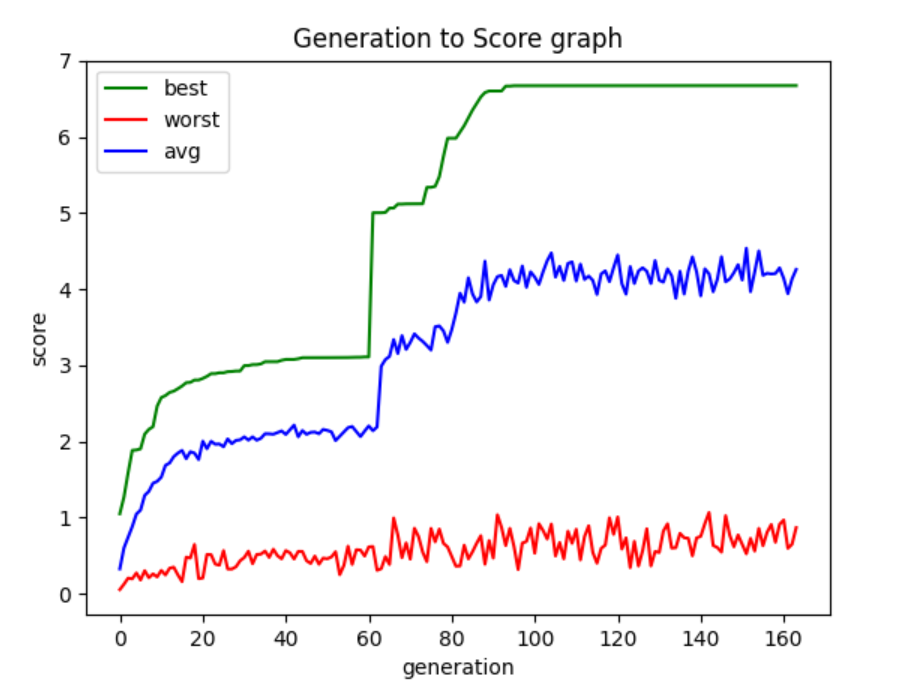
Description automatically generated with low confidence 

**DARWIN**

163 generations until convergence. Reached best score after 113 generations.

A black background with white text

Description automatically generated with low confidence

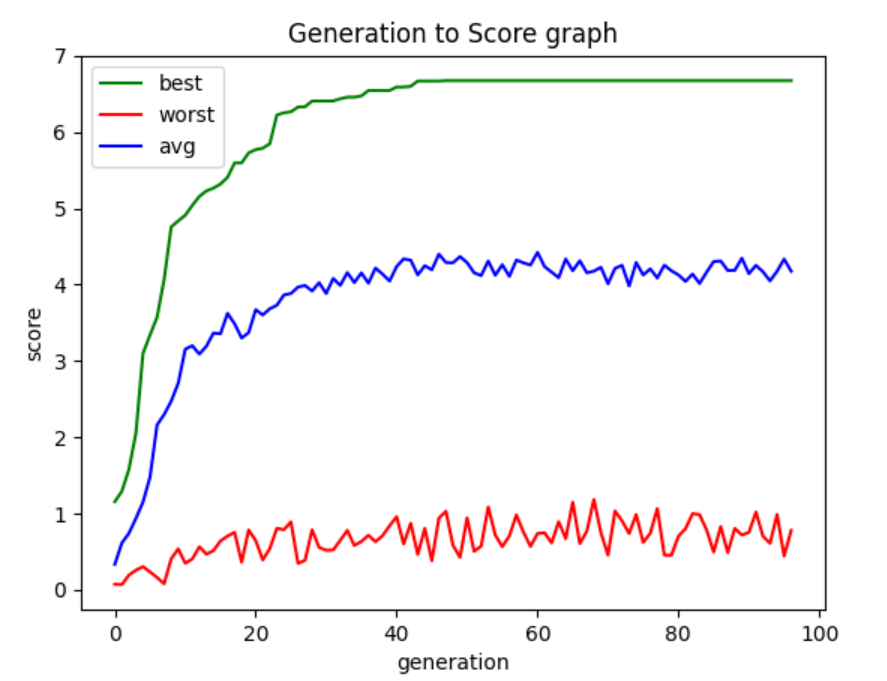


**LAMARCK**

96 generations until convergence. Reached best score after 46 generations.

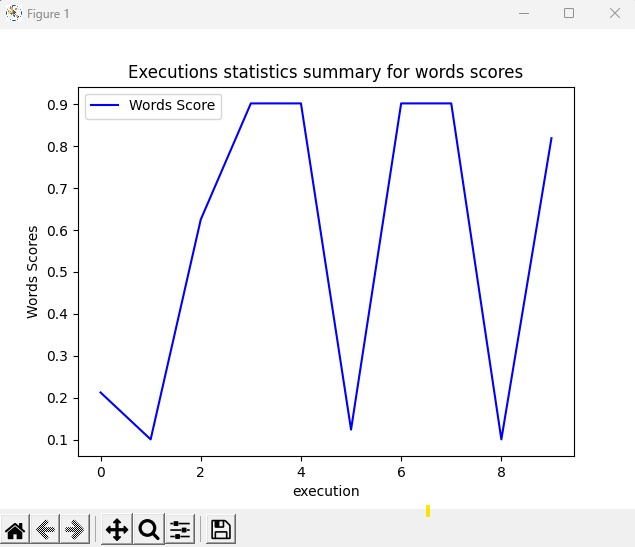
A black background with white text

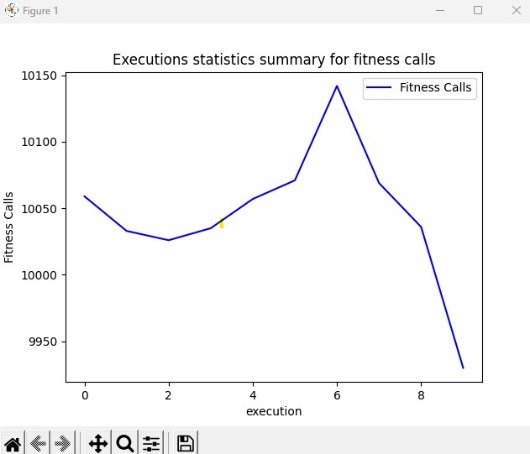
Description automatically generated with low confidence



Now we will show graphs for each algorithm, run **ten** times:

For purposes of minimizing runtime, we stopped after 50 generations.

**REGULAR**

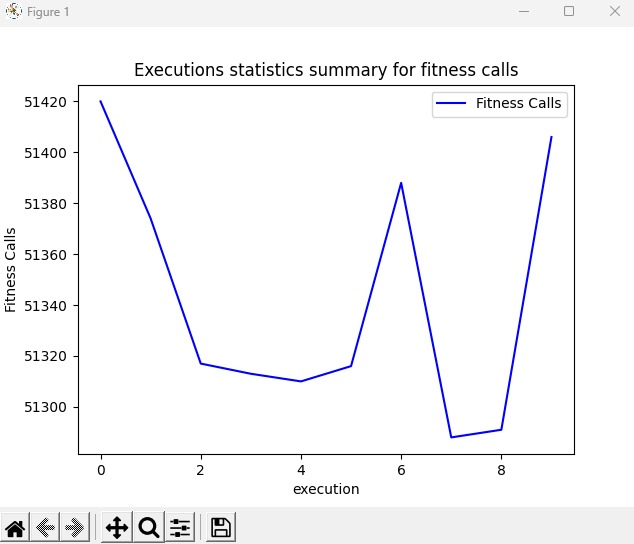


It called the fitness function on average 10,045 times (~200 population\_size \*50 gens).

The average word count was 55% accuracy, reaching peaks of 90% 5 out of the 10 times.

A screenshot of a computer screen

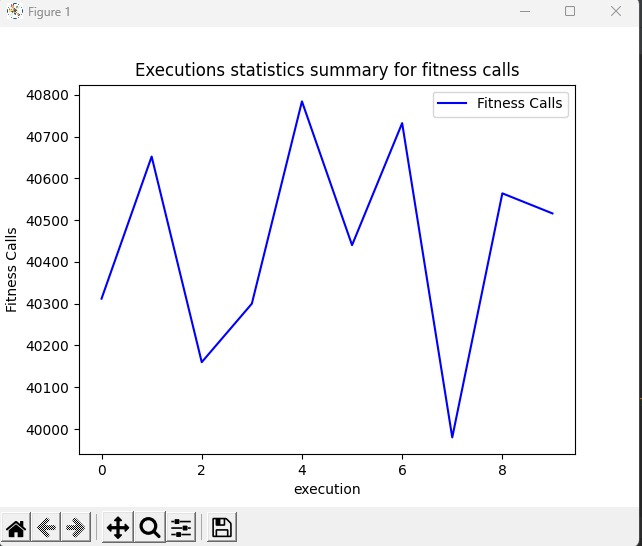
Description automatically generated with low confidence**DARWIN**



It called the fitness function on average 51,342 times (~5 times the num in regular algorithm).

The average word count reached around 64% accuracy, reaching peaks of about 90% 7 of 10 times.

**LAMARCK**



A screenshot of a computer screen

Description automatically generated with low confidence

It called the fitness function on average 40,444 times (~4 times the num in regular algorithm).

The average word count reached around 74% accuracy, reaching peaks of about 90% 8 of 10 times.

*Conclusions:*

As expected, the Lamarck algorithm performs best on average, followed by the Darwin algorithm, and then the regular.

The regular algorithm had the least calls to the fitness function (around 10,000) followed by Lamarck (around 40,000 calls) and Darwin (around 50,000 calls.)