Project 2 Report:

Matthew Kinahan, Mohammad Abbaspour, Dawson Burgess
University of Idaho
March 2025

Contents

1 Abstract

Evolution is a ubiquitous process that occurs at all levels of life. However, for evolution to occur there needs to be a process that is selected for. Selection works on gene products (proteins) and their ability to carry out their designed role. Therefore it can be said that evolution only cares about the gene sequence as long as that gene's protein provides a selective advantage. Further with codon redundancy genome mutations will only mutate respective proteins through a frame-shift mutation or a non-sense mutation, thus changing the proteins amino-acid sequence. We then sought to ask if we imposed mutation on the nucleotide level and selection on the amino-acid level across multiple generations would the alignment to ancestor be similar or different between nucleotide and amino-acid sequences and what would happen if we included horizontal gene transfer (HGT)? We included HGT in some select experiments to see how the transfer of benificial mutations to other less fit individuals impacts fitness. Our results show that at the nucleotide level, individuals diverge from ancestor less dramatically than at the amino-acid level.

2 Experiment Description

To answer this question we started with a population of 100 individuals with 99 random nucleotides (A, T, G, C). To calculate fitness we translated the nucleotide sequences to amino acid sequences and selected for individuals with a high internal hydrophobic residues and high terminal hydrophilic residues. We then changed those amino-acid sequences back to nucleotide sequences and then imposed point mutations at a 1% rate. This process was carried out for 500 generations. To align progeny to ancestor at the nucleotide level we used percent similarity at a single nucleotide level. To align amino-acid sequences of progeny to ancestor, we used the Needleman-Wunsch global alignment algorithm. To effectively compare these two methods of alignment we normalized alignment scores such that a perfect match is 1.

Two experimental conditions were simulated: (1) "mutation only," where evolution proceeds through mutation and selection, and (2) "mutation + HGT," where horizontal gene transfer (HGT) is added. Ten HGT events occurred per generation, transferring 3-base segments from high-fitness donors to low-fitness recipients. A conservation constraint was enforced on the last 20 nucleotides (positions 80–99), requiring at least 97% similarity to the ancestor in this region to simulate a conserved functional domain. Sequences failing this threshold were discarded, and the population was replenished via fitness-proportionate selection.

Each condition was run three times to account for variability, and we recorded per generation: average population fitness, number of individuals deleted due to conservation enforcement, average nucleic acid similarity, and average protein alignment score. These metrics allow us to compare the effects of HGT on adaptation and sequence divergence over time.

This experimental design allows us to test the hypothesis that HGT accelerates adaptation by facilitating the spread of beneficial genetic material, while also examining how selection for protein-level traits shapes nucleotide sequence evolution.

3 Algorithm Description

This section outlines the genetic algorithm (GA) developed to simulate the evolution of nucleic acid sequences and analyze evolution through sequence alignment. The GA evolves a population of nucleic acid sequences, translates them into amino acid sequences for fitness evaluation, and incorporates mechanisms such as mutation, horizontal gene transfer (HGT), and conservation enforcement. Evolved sequences are analyzed using global alignment algorithms to compare outcomes under different evolutionary conditions. Each component of the algorithm is described in detail below. This experiment utilized the codon table provided by GenScript for codon usage analysis [1]. Additionally, hydrophobicity scales were sourced from the CLC Genomics Workbench manual [2].

3.1 Representation

- **Encoding**: Each individual is represented as a nucleic acid sequence with a of length 99, consisting of bases A, T, G, and C. The length is divisible by 3 to ensure complete codons for amino acid translation.

- **Translation**: Sequences are translated into amino acid sequences (length 33) using a standard codon table (`CODON_TABLE`). Each three-nucleotide codon maps to an amino acid (e.g., "ATG" to "M", methionine), with unrecognized codons assigned "X". This step is important for fitness evaluation and protein-level analysis.

3.2 Population Initialization

- **Ancestor Sequence**: The initial population starts with a single, randomly generated ancestor sequence of 99 bases. The sequences have a nucleotide composition of 70% A/T (35 A's, 34 T's) and 30% G/C (15 G's, 15 C's). Bases are shuffled to randomize the sequence.

- **Initial Population**: The population consists of 100 identical copies of the ancestor sequence, providing a uniform baseline for evolution.

3.3 Fitness Function

- **Fitness Basis**: Fitness is calculated from the translated amino acid sequence, favoring hydrophobic amino acids in the middle and hydrophilic ones at the ends. This simulates a basic biochemical preference.

- **Calculation**:
  - The amino acid sequence (length 33) is split into three parts:
    1. first third (positions 0–10)
    2. middle third (positions 11–21)
    3. last third (positions 22–32).
  - Each amino acid is assigned a hydrophobicity score from a predefined table (`HYDROPHOBICITY_SCORES`), with positive values for hydrophobic amino acids (e.g., Isoleucine, I: 4.5) and negative for hydrophilic ones (e.g., Arginine, R: -4.5).
  - Fitness is computed as:

$$\begin{aligned}\text{fitness} = &\left(\frac{\sum \text{hydrophobicity scores of middle third}}{\text{length of middle third}}\right) \\ &- \left(\frac{\sum \text{hydrophobicity scores of first and last thirds}}{\text{length of first and last thirds}}\right)\end{aligned} \tag{1}$$

    where the middle third length is 11 and the combined first and last thirds length is 22.
  - Final fitness is adjusted to $\max(\text{fitness}, 0)$ to ensure non-negative values for selection.

- **Objective**: This encourages sequences with hydrophobic cores and hydrophilic termini, reflecting a biologically inspired structural goal.

3.4 Mutation

- **Mechanism**: Point mutations are independently applied to each sequence.
- **Rate**: Each base has a 0.01 (1%) mutation probability per generation.
- **Process**: If a mutation occurs (via random chance against the rate), the base is replaced with a randomly selected base (A, T, G, or C), introducing genetic diversity.

3.5 Horizontal Gene Transfer (HGT)

- **Optional Feature**: HGT is an optional operator, activated in one experimental condition to evaluate its effects.
- **Events**: 10 HGT events occur per generation when enabled.
- **Process**:
  - **Donor Selection**: The 10 highest-fitness individuals are chosen as donors.
  - **Recipient Selection**: The 10 lowest-fitness individuals are selected as recipients.
  - **Transfer**: For each pair, a random 3-base segment (codon-sized) from the donor replaces a random 3-base segment in the recipient. The start positions range from 0 to 96 to stay within the 99-base sequence.
- **Purpose**: HGT mimics lateral gene transfer. This potentially speeds up adaptation by disseminating beneficial mutations.

3.6 Conservation Enforcement

- **Conserved Region**: Positions 80–99 (length 20) must maintain at least 97% similarity to the ancestor's corresponding region.
- **Similarity Check**:
  - Global alignment (Needleman-Wunsch) is used with scoring: match = +2, mismatch = -1, gap penalty = -2.
  - Similarity is:

$$\text{similarity (\%)} = \left(\frac{\text{sum of all nucleotide comparisons}}{\text{length of region}}\right) \times 100$$

    For length 20, $\geq 97\%$ similarity requires at least 19.4 matches, allowing at most one mismatch or gap.
- **Enforcement**:
  - Post-mutation and HGT (if enabled), sequences below 97% similarity are discarded.

  - If fewer than 100 sequences remain, survivors are duplicated with fitness-proportional probabilities (adjusted by a small constant if needed to avoid zero-sum issues).

- **Outcome**: Enforcing conservation in the specified region while allowing free evolution elsewhere (positions 0–79).

3.7 Selection

- **Method**: Fitness-proportionate (roulette wheel) selection determines the next generation.
- **Process**:
  - Fitness values are adjusted to $\max(\text{fitness}, 0)$.
  - If the total fitness sum is zero, each individual gets equal probability (1/100).
  - Probabilities are:
$$P(i) = \frac{\text{fitness}_i}{\sum \text{fitness}}$$
  - 100 individuals are selected with replacement based on these probabilities.
- **Outcome**: Favors fitter individuals while preserving diversity through probabilistic choice.

3.8 Termination

- **Condition**: The simulation ends after 500 generations, allowing observable evolutionary trends.

3.9 Alignment Analysis

- **Nucleic Acid Similarity**:
  - **Algorithm**: Needleman-Wunsch global alignment compares each evolved sequence to the ancestor.
  - **Scoring**: Match = +2, mismatch = -1, gap penalty = -2.
  - **Metric**: Similarity is measured the same as in section 3.6.
  - **Purpose**: Measures nucleic acid divergence over generations.
- **Protein Alignment**:
  - **Algorithm**: Needleman-Wunsch global alignment of translated amino acid sequences against the ancestors amino acid sequence.
  - **Scoring Matrix**: BLOSUM50, with a gap penalty of -8.
  - **Metric**: Alignment score sums substitution scores and gap penalties, indicating protein-level similarity and functional evolution.
  - **Purpose**: Evaluates evolution of fitness-determining amino acid sequences.

3.10 Experimental Design

- **Conditions**: Two setups are simulated:
  1. **Mutation Only**: Evolution with mutation and selection.
  2. **Mutation + HGT**: Evolution with mutation, HGT, and selection.
- **Iterations**: Three independent runs per condition average out results.
- **Data Collection**: Per generation, the following is recorded:
  - Average population fitness.
  - Number of individuals deleted due to conservation enforcement.
  - Average nucleic acid similarity to the ancestor.
  - Average protein alignment score to the ancestors amino acid sequence.
- **Implementation**: The GA and analyses are implemented in the `EvolutionSimulator` class, with experiments run via `run_multiple_experiments`, saving data and plotting results.

4 Results

This section presents the outcomes of the computational experiments conducted, focusing on the number of deletions per generation, nucleic and protein alignment scores, and fitness over generations. All results are averaged over three runs unless otherwise specified, and visualizations are provided as figures.

4.1 Comparison of Deletions Across Experiments

Figure 1 highlights the distribution of the number of deleted individuals per generation for two experiments, labeled “avg experiment 1” and “avg experiment 2.” The box plot shows that the median number of deletions in “avg experiment 1” is roughly 12.5, with an inter-quartile range (IQR) spanning from 10 to 15. Other the other hand, “avg experiment 2” has a higher median of approximately 14.5, with an IQR from 12 to 17. Both experiments showcase similar fluctuation, with ends extending from approximately 5 to 20–22. The outliers point to occasional runs with extremely low or high deletion counts. This indicates the inclusion of HGT does not significantly alter the rate of deletions per generation.

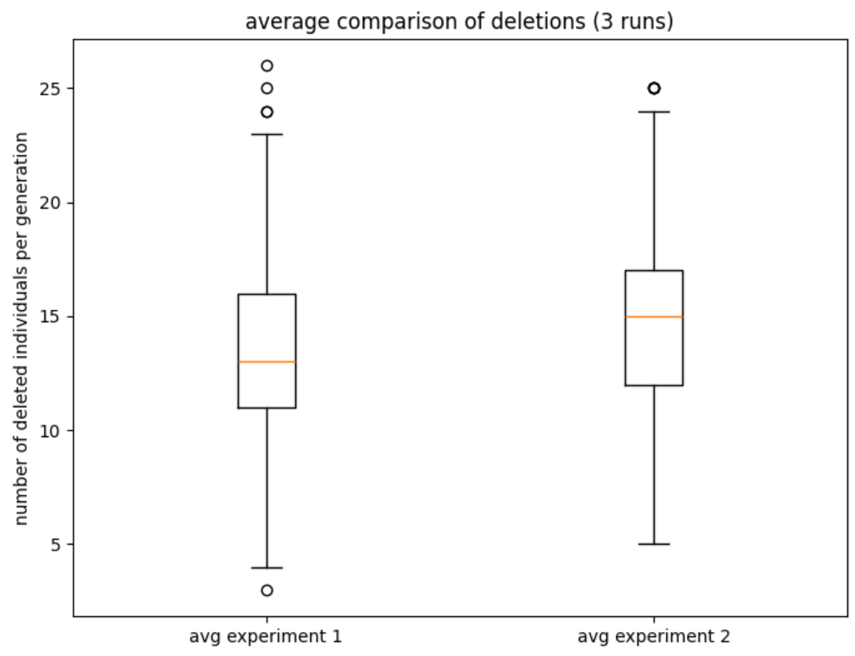

Figure 1: Box plot comparing the average number of deleted individuals per generation across three runs for "avg experiment 1" and "avg experiment 2." The y-axis represents the number of deletions, with medians marked by orange lines and outliers shown as circles.

4.2 Nucleutides and Protein Alignment Scores

Figure 2 normalized similarity of nucleic acid sequences (as a percentage) and protein alignment scores (using the BLOSUM50 matrix) over 500 generations for the averaged results of experiment 1 (mutation only) and experiment 2 (mutation + HGT), respectively. For experiment 1, nucleotide similarity stabilizes around 62.70%, while the protein alignment score drops from 200 to approximately -2.81 by the end of the simulation. For experiment 2, nucleic similarity stabilizes at 64.36%, and the protein alignment score decreases to 4.96. The higher final protein alignment score in experiment 2 suggests that HGT may help maintain more functional similarity at the protein level despite similar nucleotides divergence.

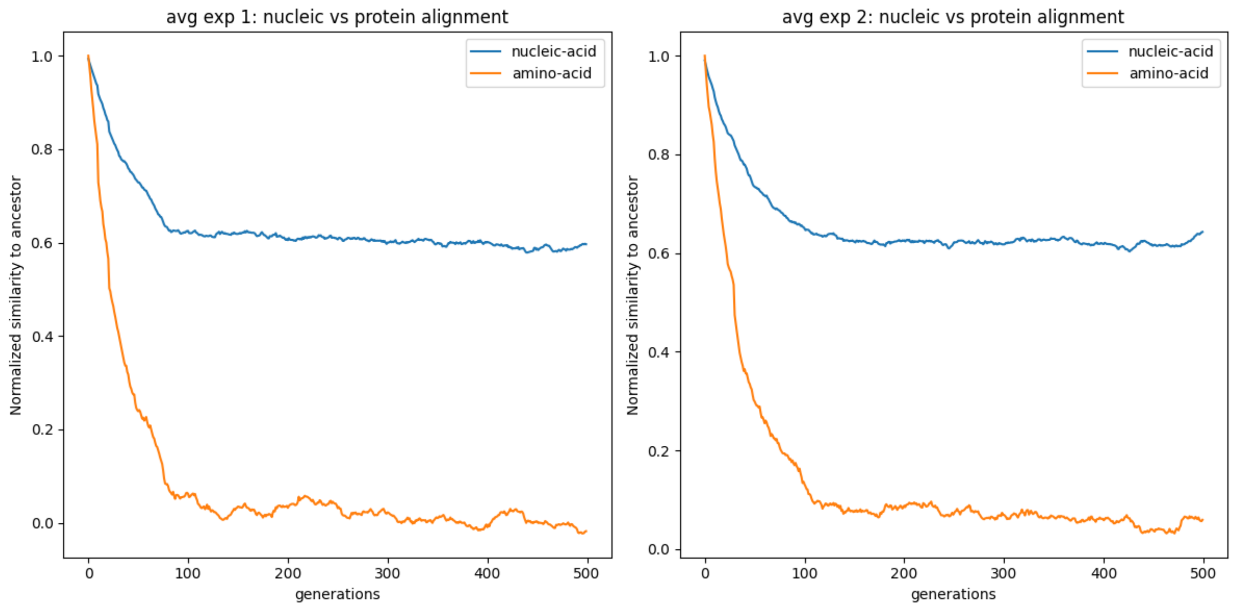

Figure 2: Line graph showing nucleotides similarity (%) and protein alignment scores over 500 generations for "avg experiment 1" and "avg experiment 2, "averaged over three runs. The blue line represents nucleotides similarity, and the orange line represents protein scores.

Figure 3 presents the nucleotides and protein alignment scores for all three runs of experiment 1 and experiment 2, respectively. In both setups, the trends are consistent across runs: nucleotides similarity stabilizes between 55–66%, while protein alignment scores show more variability, ranging from -19.58 to 20.69. The individual runs highlight the stochastic nature of the evolutionary process, with some runs maintaining higher similarity or alignment scores than others.

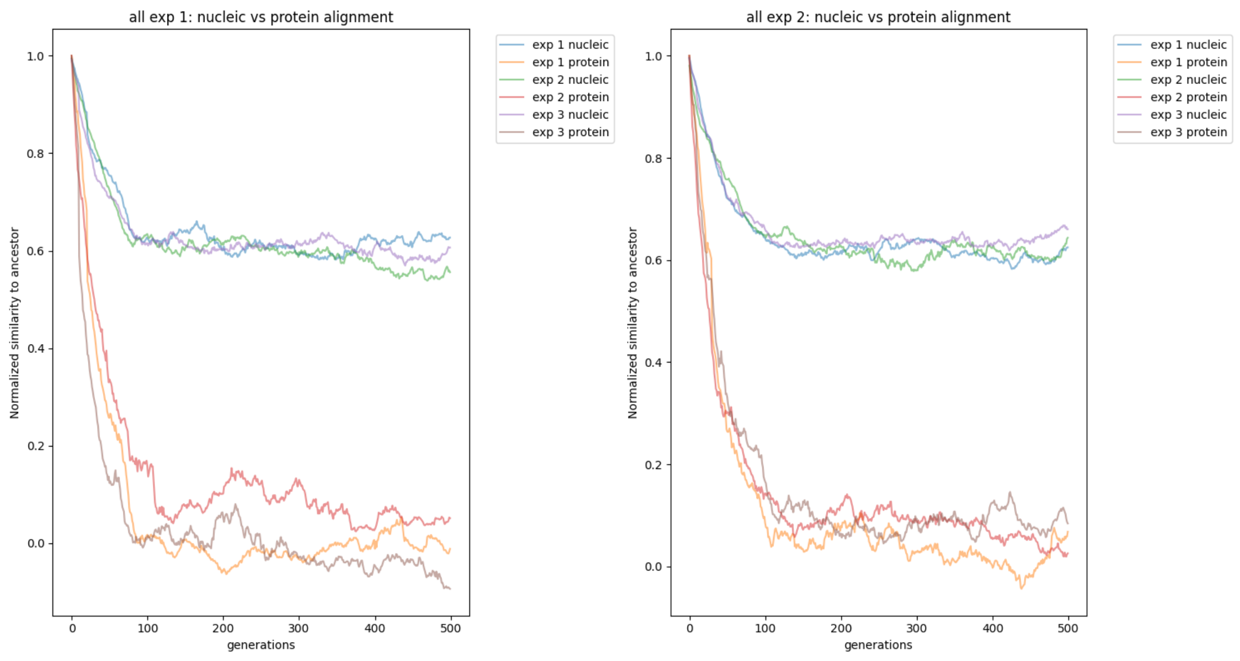

Figure 3: Line graph comparing nucleotides and protein alignment scores over 500 generations for all runs of "exp 1" and "exp 2." Colors represent different runs: blue (exp 1 nucleotides), orange (exp 1 protein), green (exp 2 nucleotides), red (exp 2 protein), purple (exp 3 nucleotides), brown (exp 3 protein).

4.3 Fitness Over Generations

Figure 4 compares the average fitness over 500 generations for the two experimental setups: "mutation only" (experiment 1) and "mutation + HGT" (experiment 2), averaged over three runs. Both setups show a rapid increase in fitness within the first 100 generations, followed by a plateau with some fluctuations. The "mutation only" setup reaches a final average fitness of 4.88, while the "mutation + HGT" setup achieves a higher final fitness of 5.27. The "mutation + HGT" setup also showcases less variability after the initial increase, suggesting that HGT may contribute to more stable fitness improvements over time.

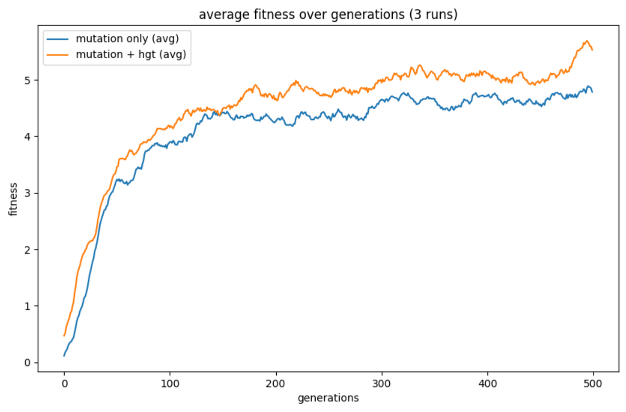

Figure 4: Line graph comparing average fitness over 500 generations for "mutation only" (blue) and "mutation + hgt" (orange), averaged over three runs.

Figure 5 displays the fitness trajectories for all runs across both experimental setups. All runs follow patterns of steep increase followed by stabilization. The "mutation + HGT" runs (red, purple, brown) generally achieve higher final fitness values (ranging from 5.22 to 6.10) compared to the "mutation only" runs (blue, orange, green), which range from 4.26 to 5.22. This supports the hypothesis that HGT accelerates adaptation by facilitating the transfer of beneficial genetic material.

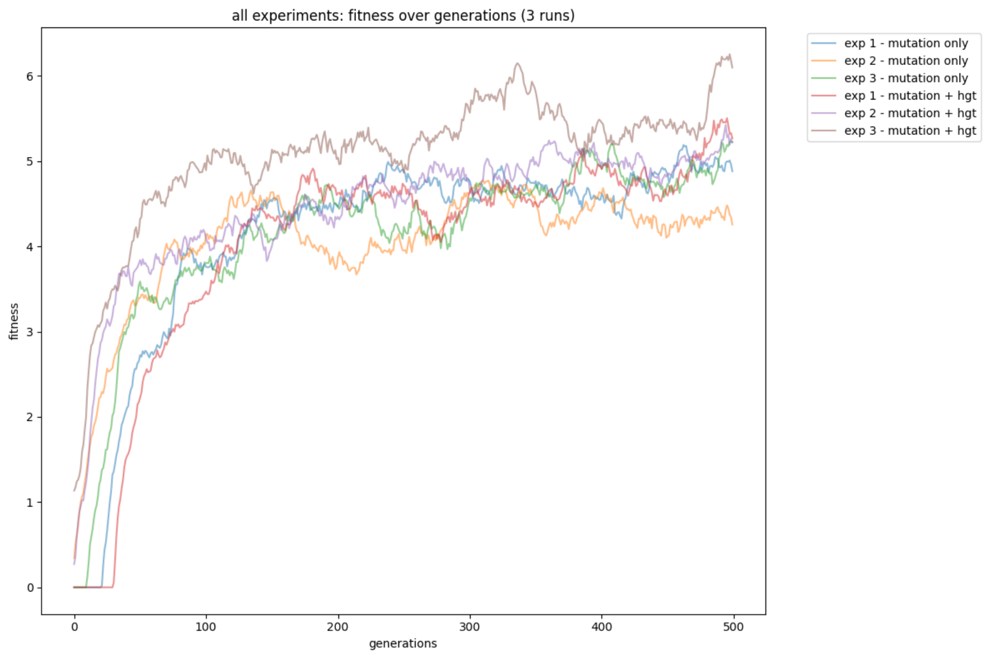

Figure 5: Line graph comparing fitness trajectories over 500 generations for all runs of "mutation only" (blue, orange, green) and "mutation + HGT" (red, purple, brown) setups.

4.4 Summary of Final Metrics

The final metrics for each experiment, averaged over three runs, are summarized as follows:

1. **Experiment 1 (mutation only)**: Final average fitness of 4.88, nucleotides similarity of 62.70%, and protein alignment score of -2.81.
2. **Experiment 1 (mutation + HGT)**: Final average fitness of 5.27, nucleotides similarity of 62.48%, and protein alignment score of 14.16.
3. **Experiment 2 (mutation only)**: Final average fitness of 4.26, nucleotides similarity of 55.59%, and protein alignment score of 11.69.
4. **Experiment 2 (mutation + HGT)**: Final average fitness of 5.22, nucleotides similarity of 64.36%, and protein alignment score of 4.96.
5. **Experiment 3 (mutation only)**: Final average fitness of 5.22, nucleotides similarity of 60.65%, and protein alignment score of -19.58.
6. **Experiment 3 (mutation + HGT)**: Final average fitness of 6.10, nucleotides similarity of 66.04%, and protein alignment score of 20.69.

These results indicate that HGT leads to higher final fitness across all experiments, with values ranging from 5.22 to 6.10 compared to 4.26 to 5.22 for mutation only. Nucleotides similarity remains somewhat stable across experiments (55.59% to 66.04%), while protein alignment scores show greater variability in the presence of HGT. The range of scores for HGT is from 4.96 to 20.69 compared to -19.58 to 11.69 for mutation only.

5 Conclusions

Our experiments provide insight into how evolution operates when selection acts on amino acids, even though mutations occur at the nucleotide level. We found that protein sequences diverged more rapidly than nucleotide sequences, which makes sense given the redundancy in the genetic code and the fact that not every nucleotide change affects protein structure or function.

One of the clearest patterns we observed was how fitness changed over time. At the beginning of each simulation, fitness increased rapidly and then gradually slowed down and leveled off. This matches what we often see in real-world evolution: Initially, beneficial mutations appear and spread rapidly, but as highly fit individuals become common, new mutations have a harder time competing. However, evolution never stops; it just becomes more gradual as adaptation continues in smaller steps. This trend supports our hypothesis and aligns with the known behavior of fitness trajectories in natural populations.

We also found that horizontal gene transfer (HGT) strongly impacted the outcome. In all experiments, HGT-positive populations reached higher fitness levels than those without it. HGT allowed beneficial mutations to spread faster, helping low-fitness individuals catch up and improving the population overall. It also helped maintain a more notable similarity at the protein level, suggesting a stronger preservation of function.

In general, our results show how selection at the protein level shapes evolution differently from selection on nucleotide sequences alone. They also highlight the important role of HGT in speeding up adaptation. These findings not only support known evolutionary patterns but also offer a model that helps us explore how genetic and functional changes play out over time in both simulated and real biological systems.

6 Running this Code

More details about the code and how to run it are available via the Github link provided: `https://github.com/dawson-b23/evolutionary-simulation-using-a-ga`

References

[1] GenScript. Codon table, 2025. Accessed: 20 March 2025.

[2] QIAGEN Bioinformatics. *Hydrophobicity Scales in CLC Genomics Workbench 6.5*, 2014. Accessed: 20 March 2025.

Appendix

Python Code for Project

```python
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt
import statistics
import os

# codon table for translating nucleotides acids to amino acids
CODON_TABLE = {
    "ATA": "I",
    "ATC": "I",
    "ATT": "I",
    "ATG": "M",
    "ACA": "T",
    "ACC": "T",
    "ACG": "T",
    "ACT": "T",
    "AAC": "N",
    "AAT": "N",
    "AAA": "K",
    "AAG": "K",
    "AGC": "S",
    "AGT": "S",
    "AGA": "R",
    "AGG": "R",
    "CTA": "L",
    "CTC": "L",
    "CTG": "L",
    "CTT": "L",
    "CCA": "P",
    "CCC": "P",
    "CCG": "P",
    "CCT": "P",
    "CAC": "H",
    "CAT": "H",
    "CAA": "Q",
    "CAG": "Q",
    "CGA": "R",
    "CGC": "R",
    "CGG": "R",
    "CGT": "R",
    "GTA": "V",
    "GTC": "V",
    "GTG": "V",
    "GTT": "V",
    "GCA": "A",
    "GCC": "A",
    "GCG": "A",
    "GCT": "A",
    "GAC": "D",
    "GAT": "D",
    "GAA": "E",
    "GAG": "E",
    "GGA": "G",
    "GGC": "G",
    "GGG": "G",
    "GGT": "G",
    "TCA": "S",
    "TCC": "S",
    "TCG": "S",
    "TCT": "S",
    "TTC": "F",
    "TTT": "F",
    "TTA": "L",
    "TTG": "L",
    "TAC": "Y",
    "TAT": "Y",
    "TAA": "*",
    "TAG": "*",
    "TGC": "C",
    "TGT": "C",
    "TGA": "*",
    "TGG": "W",
}
```

```
# hydrophobicity scores (simplified scale: positive = hydrophobic,
                                     negative = hydrophilic)
HYDROPHOBICITY_SCORES = {
    "A": 1.8,
    "R": -4.5,
    "N": -3.5,
    "D": -3.5,
    "C": 2.5,
    "Q": -3.5,
    "E": -3.5,
    "G": -0.4,
    "H": -3.2,
    "I": 4.5,
    "L": 3.8,
    "K": -3.9,
    "M": 1.9,
    "F": 2.8,
    "P": -1.6,
    "S": -0.8,
    "T": -0.7,
    "W": -0.9,
    "Y": -1.3,
    "V": 4.2,
}

amino_acids_for_blosum = {
    "A": 0,
    "R": 1,
    "N": 2,
    "D": 3,
    "C": 4,
    "Q": 5,
    "E": 6,
    "G": 7,
    "H": 8,
    "I": 9,
    "L": 10,
    "K": 11,
    "M": 12,
    "F": 13,
    "P": 14,
    "S": 15,
    "T": 16,
    "W": 17,
    "Y": 18,
    "V": 19,
    "*": 20,
}
blossum50_matrix = [
    [5, -2, -1, -2, -1, -1, -1, 0, -2, -1, -2, -1, -1, -3, -1, 1, 0, -3, -
                                          2, 0, -8],
    [-2, 7, -1, -2, -4, 1, 0, -3, 0, -4, -3, 3, -2, -3, -3, -1, -1, -3, -1
                                          , -3, -8],
[-1, -1, 7, 2, -2, 0, 0, 0, 1, -3, -4, 0, -2, -4, -2, 1, 0, -4, -2, -3
                                            , -8],
[-2, -2, 2, 8, -4, 0, 2, -1, -1, -4, -4, -1, -4, -5, -1, 0, -1, -5, -3
                                            , -4, -8],
[
    -1,
    -4,
    -2,
    -4,
    13,
    -3,
    -3,
    -3,
    -3,
    -2,
    -2,
    -3,
    -2,
    -2,
    -4,
    -1,
    -1,
    -5,
    -3,
    -1,
    -8,
],
[-1, 1, 0, 0, -3, 7, 2, -2, 1, -3, -2, 2, 0, -4, -1, 0, -1, -1, -1, -3
                                            , -8],
[-1, 0, 0, 2, -3, 2, 6, -3, 0, -4, -3, 1, -2, -3, -1, -1, -1, -3, -2,
                                            -3, -8],
[0, -3, 0, -1, -3, -2, -3, 8, -2, -4, -4, -2, -3, -4, -2, 0, -2, -3, -
                                            3, -4, -8],
[-2, 0, 1, -1, -3, 1, 0, -2, 10, -4, -3, 0, -1, -1, -2, -1, -2, -3, 2,
                                             -4, -8],
[-1, -4, -3, -4, -2, -3, -4, -4, -4, 5, 2, -3, 2, 0, -3, -3, -1, -3, -
                                            1, 4, -8],
[-2, -3, -4, -4, -2, -2, -3, -4, -3, 2, 5, -3, 3, 1, -4, -3, -1, -2, -
                                            1, 1, -8],
[-1, 3, 0, -1, -3, 2, 1, -2, 0, -3, -3, 6, -2, -4, -1, 0, -1, -3, -2,
                                            -3, -8],
[-1, -2, -2, -4, -2, 0, -2, -3, -1, 2, 3, -2, 7, 0, -3, -2, -1, -1, 0,
                                             1, -8],
[-3, -3, -4, -5, -2, -4, -3, -4, -1, 0, 1, -4, 0, 8, -4, -3, -2, 1, 4,
                                             -1, -8],
[
    -1,
    -3,
    -2,
    -1,
    -4,
    -1,
    -1,
    -2,
        -2,
        -3,
        -4,
        -1,
        -3,
        -4,
        10,
        -1,
        -1,
        -4,
        -3,
        -3,
        -8,
    ],
    [1, -1, 1, 0, -1, 0, -1, 0, -1, -3, -3, 0, -2, -3, -1, 5, 2, -4, -2, -
                                                2, -8],
    [0, -1, 0, -1, -1, -1, -1, -2, -2, -1, -1, -1, -1, -2, -1, 2, 5, -3, -
                                                2, 0, -8],
    [-3, -3, -4, -5, -5, -1, -3, -3, -3, -3, -2, -3, -1, 1, -4, -4, -3, 15
                                                , 2, -3, -8],
    [-2, -1, -2, -3, -3, -1, -2, -3, 2, -1, -1, -2, 0, 4, -3, -2, -2, 2, 8
                                                , -1, -8],
    [0, -3, -3, -4, -1, -3, -3, -4, -4, 4, 1, -3, 1, -1, -3, -2, 0, -3, -1
                                                , 5, -8],
    [-8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -8, -
                                                8, -8, -8, 8],
]


class EvolutionSimulator:
    def __init__(self):
        # simulation parameters
        self.population_size = 100
        self.sequence_length = 99  # must be divisible by 3 for codons
        self.mutation_rate = 0.01
        self.horizontal_gene_transfer_events = 10
        self.region_start = 80
        self.region_end = 99
        self.similarity_threshold = 97
        self.maximum_generations = 500
        self.gap = -8

        # initialize ancestor and population
        self.ancestor_nucleic_acid = self.generate_ancestor_nucleic_acid()
        self.population_nucleic_acid = self.generate_population()

    def generate_ancestor_nucleic_acid(self):
        # generate random ancestor sequence (70% a/t, 30% g/c)
        bases = ["A"] * 35 + ["T"] * 34 + ["G"] * 15 + ["C"] * 15
        random.shuffle(bases)
        return "".join(bases)

    def generate_population(self):
        # generate initial population of identical sequences
        return [self.ancestor_nucleic_acid[:] for _ in range(self.
                                                 population_size)]

    def translate_to_amino_acid(self, nucleic_acid_sequence):
        # translate nucleic acid sequence to amino acid sequence using
                                                 codons
        amino_acid_sequence = ""
        for i in range(0, len(nucleic_acid_sequence), 3):
            codon = nucleic_acid_sequence[i : i + 3]
            if len(codon) == 3:
                amino_acid_sequence += CODON_TABLE.get(codon, "X")
        return amino_acid_sequence

    def mutate(self, nucleic_acid_sequence):
        # mutate a sequence based on mutation rate
        sequence_list = list(nucleic_acid_sequence)
        for i in range(len(sequence_list)):
            if random.random() < self.mutation_rate:
                sequence_list[i] = random.choice(["A", "T", "G", "C"])
        return "".join(sequence_list)

    def global_alignment(self, sequence1, sequence2):
        # perform global alignment using needleman-wunsch algorithm with
                                                 blosum50-like scoring
        match_score = 2
        mismatch_penalty = -1
        gap_penalty = -2

        length1, length2 = len(sequence1), len(sequence2)
        dynamic_programming_matrix = np.zeros((length1 + 1, length2 + 1))

        # initialize first row and column with gap penalties
        for i in range(length1 + 1):
            dynamic_programming_matrix[i][0] = i * gap_penalty
        for j in range(length2 + 1):
            dynamic_programming_matrix[0][j] = j * gap_penalty

        # fill the matrix
        for i in range(1, length1 + 1):
            for j in range(1, length2 + 1):
                match = dynamic_programming_matrix[i - 1][j - 1] + (
                    match_score
                    if sequence1[i - 1] == sequence2[j - 1]
                    else mismatch_penalty
                )
                up = dynamic_programming_matrix[i - 1][j] + gap_penalty
                left = dynamic_programming_matrix[i][j - 1] + gap_penalty
                dynamic_programming_matrix[i][j] = max(match, up, left)

        # compute similarity
        matches = 0
        i, j = length1, length2
        while i > 0 and j > 0:
            if sequence1[i - 1] == sequence2[j - 1]:
                matches += 1
            if dynamic_programming_matrix[i][j] ==
                                                   dynamic_programming_matrix
                                                   [i - 1][
                j - 1
            ] + (
                match_score
                if sequence1[i - 1] == sequence2[j - 1]
                else mismatch_penalty
            ):
                i -= 1
                j -= 1
            elif (
                dynamic_programming_matrix[i][j]
                == dynamic_programming_matrix[i - 1][j] + gap_penalty
            ):
                i -= 1
            else:
                j -= 1

        return (matches / length1) * 100

    def enforce_conservation(self):
        # enforce 97% similarity in conserved region (80-99)
        ancestor_region = self.ancestor_nucleic_acid[
            self.region_start : self.region_end
        ]
        survivors = [
            seq
            for seq in self.population_nucleic_acid
            if self.global_alignment(
                seq[self.region_start : self.region_end], ancestor_region
            )
            >= self.similarity_threshold
        ]
        deleted_count = self.population_size - len(survivors)
        while len(survivors) < self.population_size:
            fitness_values = np.array(
                [self.calculate_fitness(seq) for seq in survivors], dtype=
                                                       np.float64
            )
            fitness_values = np.maximum(fitness_values, 0)
            if fitness_values.sum() == 0:
                fitness_values += 1e-6
            probabilities = fitness_values / fitness_values.sum()
            selected_index = np.random.choice(len(survivors), p=
                                                   probabilities)
            survivors.append(survivors[selected_index])
        self.population_nucleic_acid = survivors
        return deleted_count

    def calculate_fitness(self, nucleic_acid_sequence):
        # fitness based on hydrophilic ends and hydrophobic middle using
                                                translated sequence
        amino_acid_sequence = self.translate_to_amino_acid(
                                            nucleic_acid_sequence)
        length = len(amino_acid_sequence)
        third = length // 3

        ends = amino_acid_sequence[:third] + amino_acid_sequence[-third:]
        middle = amino_acid_sequence[third:-third]

        ends_score = sum(HYDROPHOBICITY_SCORES.get(aa, 0) for aa in ends)
        middle_score = sum(HYDROPHOBICITY_SCORES.get(aa, 0) for aa in
                                            middle)

        fitness = (middle_score / len(middle)) - (ends_score / len(ends))
        return max(fitness, 0)

    def select_next_generation(self):
        # select next generation using fitness-proportionate selection
        fitness_values = np.array(
            [self.calculate_fitness(seq) for seq in self.
                                                population_nucleic_acid]
        )
        # ensure no negative fitness values
        fitness_values = np.maximum(fitness_values, 0)
        # handle case where all fitness values are zero
        if fitness_values.sum() == 0:
            fitness_values = np.ones_like(
                fitness_values
            )  # equal probability if all zero
        probabilities = fitness_values / fitness_values.sum()
        selected_indices = np.random.choice(
            len(self.population_nucleic_acid),
            size=self.population_size,
            p=probabilities,
        )
        self.population_nucleic_acid = [
            self.population_nucleic_acid[i] for i in selected_indices
        ]

    def perform_horizontal_gene_transfer(self):
        # transfer random 3-base segment from high-fitness donors to low-
                                            fitness recipients
        fitness_values = [
            self.calculate_fitness(seq) for seq in self.
                                                population_nucleic_acid
        ]
        donor_indices = np.argpartition(
            fitness_values, -self.horizontal_gene_transfer_events
        )[-self.horizontal_gene_transfer_events :]
        donors = [self.population_nucleic_acid[i] for i in donor_indices]

        recipient_indices = np.argpartition(
            fitness_values, self.horizontal_gene_transfer_events
        )[: self.horizontal_gene_transfer_events]
        recipients = [self.population_nucleic_acid[i] for i in
                                                      recipient_indices]

        for donor, recipient in zip(donors, recipients):
            start_position = random.randint(0, self.sequence_length - 3)
            segment = donor[start_position : start_position + 3]
            recipient_start = random.randint(0, self.sequence_length - 3)
            modified_recipient = (
                recipient[:recipient_start] + segment + recipient[
                                                     recipient_start + 3 :
                                                     ]
            )
            self.population_nucleic_acid[
                self.population_nucleic_acid.index(recipient)
            ] = modified_recipient

    def run_experiment(self, use_horizontal_gene_transfer=False):
        # run simulation with or without horizontal gene transfer
        print(
            f"starting experiment: {'mutation + hgt' if
                                             use_horizontal_gene_transfer
                                              else 'mutation only'}"
        )
        generation = 0
        fitness_progress = []
        deleted_individuals = []
        nucleic_acid_sequences = []

        while generation < self.maximum_generations:
            generation += 1

            self.population_nucleic_acid = [
                self.mutate(seq) for seq in self.population_nucleic_acid
            ]
            if use_horizontal_gene_transfer:
                self.perform_horizontal_gene_transfer()

            deleted_count = self.enforce_conservation()
            self.select_next_generation()

            avg_fitness = np.mean(
                [self.calculate_fitness(seq) for seq in self.
                                                     population_nucleic_acid
                                                     ]
            )
            fitness_progress.append(avg_fitness)
            deleted_individuals.append(deleted_count)
            nucleic_acid_sequences.append(self.population_nucleic_acid.
                                             copy())

            if generation % 50 == 0:
                print(
                    f"generation {generation}: avg fitness = {avg_fitness
                                                            :.2f}, deleted =
                                                                {deleted_count}"
                )
                print(f"  sample nucleic acid: {self.
                                                            population_nucleic_acid
                                                            [0]}")

        avg_fitness = statistics.mean(fitness_progress)
        print(
            f"experiment completed: {generation} generations, final avg
                                                        fitness = {avg_fitness:.
                                                        2f}"
        )
        return generation, fitness_progress, deleted_individuals,
                                                    nucleic_acid_sequences

    def analyze_evolution(self, nucleic_acid_sequences):
        # analyze evolutionary process using global alignment on nucleic
                                                    acids
        alignments_over_time = []
        for gen_idx, generation_sequences in enumerate(
                                                    nucleic_acid_sequences):
            avg_similarity = np.mean(
                [
                    self.global_alignment(seq, self.ancestor_nucleic_acid)
                    for seq in generation_sequences
                ]
            )
            alignments_over_time.append(avg_similarity)
            if (gen_idx + 1) % 50 == 0:
                print(
                    f"analysis at generation {gen_idx + 1}: avg similarity
                                                                 to ancestor = {
                                                                avg_similarity:.
                                                                2f}%"
                )
        return alignments_over_time

    def protein_alignment(self, nucleic_acid_sequences):
        # print ancestor sequence
        # print("Ancestor")
        # print(self.ancestor_nucleic_acid)
        # translate ancestor to amino acid
        anc_translated = self.translate_to_amino_acid(self.
                                                    ancestor_nucleic_acid)
        final_translated = []
        # translate all sequences in each generation
        for gen in nucleic_acid_sequences:
            gen_translated = []
            for member in gen:
                gen_translated.append(self.translate_to_amino_acid(member)
                                                            )
            final_translated.append(gen_translated)

        # begin alignments
        score_to_anc = []
        for g in final_translated:
            gen_list = []
            for evo in g:
                # generate matrix
                score = [
                    [0 for x in range(0, len(anc_translated) + 1)]
                    for y in range(0, len(evo) + 1)
                ]
                # initialize first row and column with gap penalties
                for c in range(0, len(anc_translated) + 1):
                    score[0][c] = c * self.gap
                for r in range(0, len(evo) + 1):
                    score[r][0] = r * self.gap

                # use nw algo for alignment
                for c in range(1, len(anc_translated) + 1):
                    for r in range(1, len(evo) + 1):
                        scoreD = (
                            score[r - 1][c - 1]
                            + blossum50_matrix[amino_acids_for_blosum[evo[
                                                                       r - 1]]][
                                amino_acids_for_blosum[anc_translated[c -
                                                                         1]]
                            ]
                        )
                        scoreV = score[r - 1][c] + self.gap
                        scoreH = score[r][c - 1] + self.gap
                        score[r][c] = max(scoreD, scoreV, scoreH)
                gen_list.append(score[-1][-1])
            score_to_anc.append(gen_list)

        # calculate average scores per generation
        avg_scores = [np.mean(scores) for scores in score_to_anc]

        # return all three values
        return score_to_anc, final_translated, avg_scores


def run_multiple_experiments(iterations=3):
    # store results for averaging
    all_exp1_results = []
    all_exp2_results = []

    # run specified number of iterations
    for i in range(1, iterations + 1):
        exp_dir = f"experiment_{i}"
        os.makedirs(exp_dir, exist_ok=True)

        print(f"--- starting experiment {i} ---")

        # experiment 1: mutation only
        simulator1 = EvolutionSimulator()
        gen1, fitness1, deleted1, sequences1 = simulator1.run_experiment(
        use_horizontal_gene_transfer=False
    )
    protein_scores1, translated_seqs1, avg_protein_scores1 = (
        simulator1.protein_alignment(sequences1)
    )
    alignment_progress1 = simulator1.analyze_evolution(sequences1)

    # experiment 2: mutation + hgt
    simulator2 = EvolutionSimulator()
    gen2, fitness2, deleted2, sequences2 = simulator2.run_experiment(
        use_horizontal_gene_transfer=True
    )
    protein_scores2, translated_seqs2, avg_protein_scores2 = (
        simulator2.protein_alignment(sequences2)
    )
    alignment_progress2 = simulator2.analyze_evolution(sequences2)
    print('WHAT IS PASSED')
    print(alignment_progress2)
    print(alignment_progress1)

    print(protein_scores1)
    print(protein_scores2)
    # create dataframes for this experiment
    df_exp1 = pd.DataFrame(
        {
            "generation": list(range(len(fitness1))),
            "fitness": fitness1,
            "deleted": deleted1,
            "nucleic_similarity":  [(x/100) for x in
                                        alignment_progress1],
            "avg_protein_score": [(t/max(avg_protein_scores1)) for t
                                        in
                                        avg_protein_scores1],
            "nucleic_acid_sequences": sequences1,
            "protein_sequences": translated_seqs1,
        }
    )
    df_exp2 = pd.DataFrame(
        {
            "generation": list(range(len(fitness2))),
            "fitness": fitness2,
            "deleted": deleted2,
            "nucleic_similarity": [(b/100) for b in
                                        alignment_progress2],
            "avg_protein_score": [(u/max(avg_protein_scores2)) for u
                                        in
                                        avg_protein_scores2],
            "nucleic_acid_sequences": sequences2,
            "protein_sequences": translated_seqs2,
        }
    )

    # save individual experiment results
    df_exp1.to_csv(f"{exp_dir}/experiment1_data.csv", index=False)
```

```
df_exp2.to_csv(f"{exp_dir}/experiment2_data.csv", index=False)

# store results for averaging
all_exp1_results.append(df_exp1)
all_exp2_results.append(df_exp2)

# plot individual experiment results
plt.figure(figsize=(10, 6))
plt.plot(df_exp1["generation"], df_exp1["fitness"], label="
                                mutation only")
plt.plot(df_exp2["generation"], df_exp2["fitness"], label="
                                mutation + hgt")
plt.xlabel("generations")
plt.ylabel("fitness")
plt.title(f"experiment {i}: fitness over generations")
plt.legend()
plt.savefig(f"{exp_dir}/fitness_over_generations.png")
plt.close()

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(
    df_exp1["generation"], df_exp1["nucleic_similarity"], label="
                                    nucleic (%)"
)
plt.plot(
    df_exp1["generation"], df_exp1["avg_protein_score"], label="
                                    protein (score)"
)
plt.xlabel("generations")
plt.ylabel("Normalized similarity to ancestor")
plt.title(f"exp {i}-1: nucleic vs protein alignment")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(
    df_exp2["generation"], df_exp2["nucleic_similarity"], label="
                                    nucleic-acid"
)
plt.plot(
    df_exp2["generation"], df_exp2["avg_protein_score"], label="
                                    amino-acid"
)
plt.xlabel("generations")
plt.ylabel("similarity (%) / alignment score")
plt.title(f"exp {i}-2: nucleic vs protein alignment")
plt.legend()
plt.tight_layout()
plt.savefig(f"{exp_dir}/protein_vs_nucleic_alignment.png")
plt.close()

plt.figure(figsize=(8, 6))
plt.boxplot([deleted1, deleted2], labels=["experiment 1", "
                                experiment 2"])
    plt.ylabel("number of deleted individuals per generation")
    plt.title(f"experiment {i}: comparison of deletions")
    plt.savefig(f"{exp_dir}/deletions_boxplot.png")
    plt.close()

    # print summary
    print(f"--- experiment {i} summary ---")
    print("mutation only:")
    print(f"  generations: {gen1}")
    print(f"  final avg fitness: {fitness1[-1]:.2f}")
    print(f"  final nucleic similarity: {alignment_progress1[-1]:.2f}%
                                     ")
    print(f"  final avg protein alignment score: {avg_protein_scores1
                                     [-1]:.2f}")
    print("mutation + hgt:")
    print(f"  generations: {gen2}")
    print(f"  final avg fitness: {fitness2[-1]:.2f}")
    print(f"  final nucleic similarity: {alignment_progress2[-1]:.2f}%
                                     ")
    print(f"  final avg protein alignment score: {avg_protein_scores2
                                     [-1]:.2f}")
    print("******************************")

# calculate averages across all experiments
avg_exp1 = pd.DataFrame(
    {
        "generation": range(500),  # fixed at 500
        "fitness": np.mean([df["fitness"] for df in all_exp1_results],
                                     axis=0),
        "deleted": np.mean([df["deleted"] for df in all_exp1_results],
                                     axis=0),
        "nucleic_similarity": np.mean(
            [df["nucleic_similarity"] for df in all_exp1_results],
                                     axis=0
        ),
        "avg_protein_score": np.mean(
            [df["avg_protein_score"] for df in all_exp1_results], axis
                                     =0
        ),
    }
)
avg_exp2 = pd.DataFrame(
    {
        "generation": range(500),  # fixed at 500
        "fitness": np.mean([df["fitness"] for df in all_exp2_results],
                                     axis=0),
        "deleted": np.mean([df["deleted"] for df in all_exp2_results],
                                     axis=0),
        "nucleic_similarity": np.mean(
            [df["nucleic_similarity"] for df in all_exp2_results],
                                     axis=0
        ),
        "avg_protein_score": np.mean(
            [df["avg_protein_score"] for df in all_exp2_results], axis
                                                            =0
        ),
    }
)

# save averaged results in current directory
avg_exp1.to_csv("avg_experiment1_data.csv", index=False)
avg_exp2.to_csv("avg_experiment2_data.csv", index=False)

# plot averaged results
plt.figure(figsize=(10, 6))
plt.plot(avg_exp1["generation"], avg_exp1["fitness"], label="mutation
                                                only (avg)")
plt.plot(avg_exp2["generation"], avg_exp2["fitness"], label="mutation
                                                + hgt (avg)")
plt.xlabel("generations")
plt.ylabel("fitness")
plt.title(f"average fitness over generations ({iterations} runs)")
plt.legend()
plt.savefig("avg_fitness_over_generations.png")
plt.close()

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(
    avg_exp1["generation"], avg_exp1["nucleic_similarity"], label="
                                                    nucleic-acid"
)
plt.plot(
    avg_exp1["generation"], avg_exp1["avg_protein_score"], label="
                                                    amino-acid"
)
plt.xlabel("generations")
plt.ylabel("Normalized similarity to ancestor")
plt.title("avg exp 1: nucleic vs protein alignment")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(
    avg_exp2["generation"], avg_exp2["nucleic_similarity"], label="
                                                    nucleic-acid"
)
plt.plot(
    avg_exp2["generation"], avg_exp2["avg_protein_score"], label="
                                                    amino-acid"
)
plt.xlabel("generations")
plt.ylabel("Normalized similarity to ancestor")
plt.title("avg exp 2: nucleic vs protein alignment")
plt.legend()
plt.tight_layout()
plt.savefig("avg_protein_vs_nucleic_alignment.png")
plt.close()
```

```
plt.figure(figsize=(8, 6))
plt.boxplot(
    [
        np.concatenate([df["deleted"] for df in all_exp1_results]),
        np.concatenate([df["deleted"] for df in all_exp2_results]),
    ],
    labels=["avg experiment 1", "avg experiment 2"],
)
plt.ylabel("number of deleted individuals per generation")
plt.title(f"average comparison of deletions ({iterations} runs)")
plt.savefig("avg_deletions_boxplot.png")
plt.close()

# fitness plot for all experiments
plt.figure(figsize=(12, 8))
for i, df in enumerate(all_exp1_results, 1):
    plt.plot(
        df["generation"], df["fitness"], label=f"exp {i} - mutation
                                                    only", alpha=0.5
    )
for i, df in enumerate(all_exp2_results, 1):
    plt.plot(
        df["generation"],
        df["fitness"],
        label=f"exp {i} - mutation + hgt",
        alpha=0.5,
    )
plt.xlabel("generations")
plt.ylabel("fitness")
plt.title(f"all experiments: fitness over generations ({iterations}
                                                runs)")
plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
plt.tight_layout()
plt.savefig("all_experiments_fitness.png")
plt.close()

# nucleic vs protein alignment for all experiments
plt.figure(figsize=(15, 8))
plt.subplot(1, 2, 1)
for i, df in enumerate(all_exp1_results, 1):
    plt.plot(
        df["generation"],
        df["nucleic_similarity"],
        label=f"exp {i} nucleic",
        alpha=0.5,
    )
    plt.plot(
        df["generation"],
        df["avg_protein_score"],
        label=f"exp {i} protein",
        alpha=0.5,
    )
plt.xlabel("generations")
```

```
    plt.ylabel("Normalized similarity to ancestor")
    plt.title("all exp 1: nucleic vs protein alignment")
    plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")

    plt.subplot(1, 2, 2)
    for i, df in enumerate(all_exp2_results, 1):
        plt.plot(
            df["generation"],
            df["nucleic_similarity"],
            label=f"exp {i} nucleic",
            alpha=0.5,
        )
        plt.plot(
            df["generation"],
            df["avg_protein_score"],
            label=f"exp {i} protein",
            alpha=0.5,
        )
    plt.xlabel("generations")
    plt.ylabel("Normalized similarity to ancestor")
    plt.title("all exp 2: nucleic vs protein alignment")
    plt.legend(bbox_to_anchor=(1.05, 1), loc="upper left")
    plt.tight_layout()
    plt.savefig("all_experiments_alignment.png")
    plt.close()


def main():
    # run multiple experiments with default 3 iterations
    run_multiple_experiments(iterations=3)


if __name__ == "__main__":
    main()
```