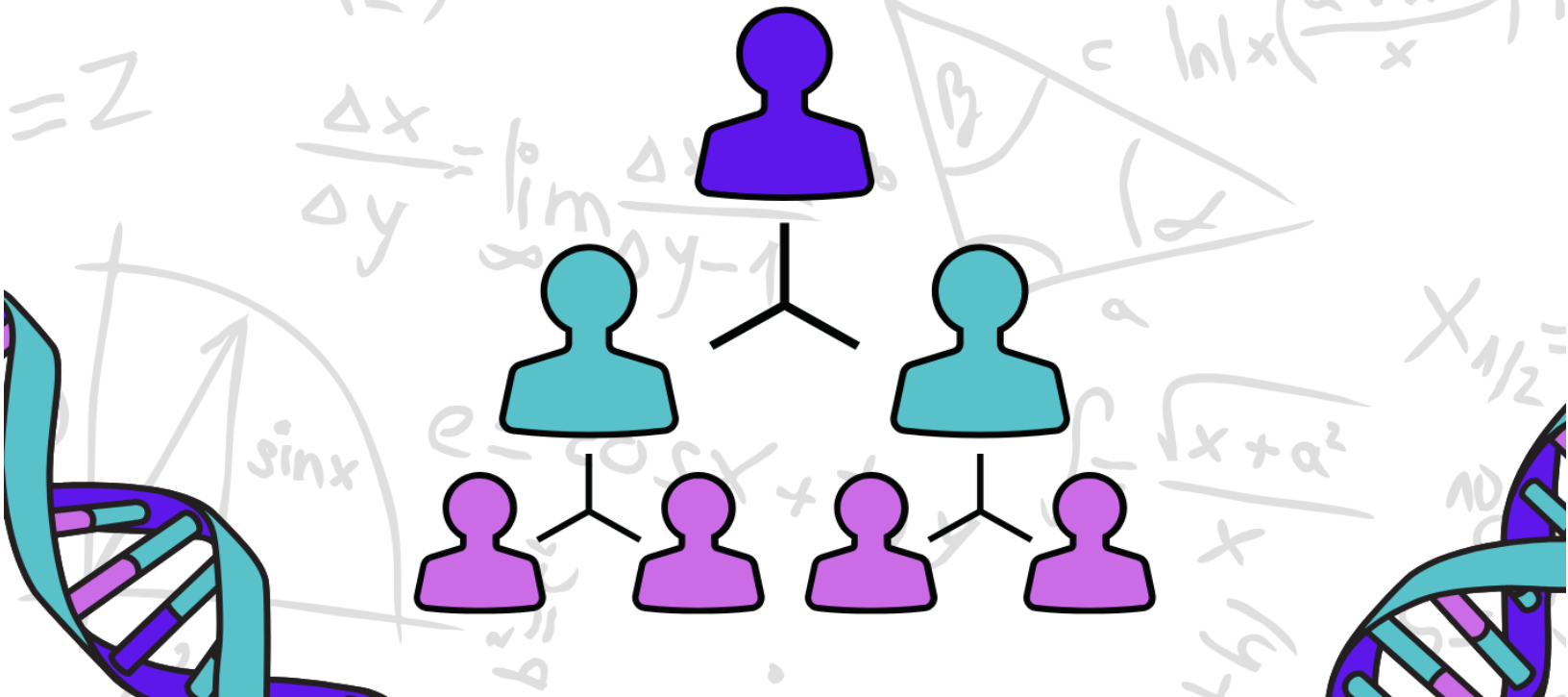
Polynomial Equation Solver By Genetic Algorithm

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0$$

Group Members:

- Abdul Moiz Mateen CS-22136
- Farzam Nasir CS-22137
- Hamza Ali CS-22146

Submitted To Dr Hameeza Ahmed



PROJECT DESCRIPTION:

The project uses a Genetic Algorithm to solve equations of any degree, to provide a strong solution toward getting roots or approximate solutions for complex mathematical problems. Analytical or numerical methods often have considerable problems in the solution of degrees that are higher or non-linear since they use a high amount of computation or tend to diverge. This genetic algorithm uses evolutionary principles inspired from natural selection effectively to overcome these problems.

The algorithm encodes potential solutions as chromosomes and iteratively evolves through such operations as selection, crossover, and mutation. In order to find the optimal or nearly optimal solution, a well-defined fitness function is developed. The adaptability of this algorithm and the simplicity allow it to be scalable to solve equations of different characteristics within computational problem-solving.

This project demonstrates the power of GAs in solving non-linear optimization problems and makes visible their potential applications in mathematics, engineering, and other fields.

Genetic Algorithm:

- **Step 1 – Initialization**
  The parameters for the algorithm are defined as:

  populationSize = 10
  The size of the population i.e. no. of chromosomes
  generations = 100
  Number of generations the algorithm will evolved through
  pc = 0.7
  The **crossover probability**, the likelihood of combining two parents to create an offspring.
  mc = 0.02
  The **mutation probability**, chances of random alterations in an offspring to introduce diversity
  lower_bound, upper_bound = -10, 10
  The search space for the algorithm

  The algorithm will work exclusively in the defined range, making it incapable of finding the roots outside the interval [lower_bound, upper_bound]. Additionally tuning the population size, generations, probabilities (pc and mc) directly impacts the algorithm's performance by striking a balance between exploration and exploitation. These values are determined through multiple trials and may require further fine-tuning for specific cases to achieve optimal performance.

- **Step 2 – Fitness Function Definition**
  When evaluating potential roots, substituting them into the equation yields result. If the result is zero, the candidate is a root of the equation. The closer the result is to zero, the higher the likelihood of it being a root. Based on this principle, the fitness function is defined as:

$$f(x) = \frac{1}{1 + |x|}$$

  The function outputs the values in the range of (0-1). With 1 representing that the result is 0, confirming that x is the root of the equation. 0 representing that the result deviates significantly from zero, meaning x is not a root of the equation. The intermediate values between 0 and 1 represent varying degrees of likelihood that x is close to a root.
  This fitness function offers a smooth, normalized metric to help the algorithm identify and refine potential roots.

- **Step 3 – Generating Initial Population**

  Before the run (iterative process) initial population is defined. The population is generated randomly, this introduces diversity among candidate solutions (chromosomes) and ensures that the population covers a broad range of the search space. The initial population is established as follows:

  population = np.random.uniform(-10, 10, populationSize)

  The function np.random.uniform generates an array of random floating-point numbers within the range [-10, 10], with the size set to populationSize (in this case, 10).

- **Step 4 – Iteration Process**

  The fitness value of each chromosome in the population is evaluated using the fitness function.

  fitnessValue = np.array([fitnessFunction(ind) for ind in population])

  Selection:

  The selection of parent chromosomes for crossover is based on their fitness values. First, the selection probabilities (fitness ratios) are calculated from the fitness values. Then, the parents are chosen using roulette wheel selection, where the probability of selection is proportional to the fitness value of each chromosome.

  ```
  probabilities = fitnessValue / fitnessValue.sum()
  selectedParent = np.random.choice(range(populationSize), size=populationSize,
  p=probabilities)
  selectedPopulation = population[selectedParent]
  ```

  Once the parents are selected, the crossover operation is applied.

  Crossover:

  From the population of randomly selected parents, pairs of parents are chosen iteratively for the crossover operation. A crossover point is randomly selected where the two parents will break, and the chromosome segments after the crossover point are exchanged. The crossover operation is performed based on the crossover probability pc. This means that crossover does not happen every time—if it doesn't occur, the parents are simply cloned, and they become part of the new generation. The crossover operation is implemented as follows:

  ```
  offspring = []
  for i in range(0, populationSize, 2):
  if np.random.rand() < pc:
  crossover_point = np.random.rand()
      parent1, parent2 = selectedPopulation[i], selectedPopulation[i+1]
      child1 = crossover_point * parent1 + (1 - crossover_point) * parent2
  ```

```
child2 = crossover_point * parent2 + (1 - crossover_point) * parent1
        offspring.extend([child1, child2])
else:
            offspring.extend([selectedPopulation[i], selectedPopulation[i+1]])
```

The function np.random.rand generates a random floating point number in the range of 0 and 1 with an equal likelihood of every value in this range. If the number is less than the crossover probability **pc**, the crossover operation is carried out otherwise no crossover operation happens and cloning takes place. This ensures that the crossover occurs based on the defined probability, controlling how often new genetic combinations are introduced.

Mutation:

After the crossover operation, the mutation step is performed. In this step, random mutations are introduced to the offspring by adding a random value to their chromosomes, which promotes diversity in the population. The mutation process is implemented as follows:

```
for i in range(len(offspring)):
    if np.random.rand() < mc:
offspring[i] += np.random.uniform(-0.5, 0.5)  # Random mutation step
offspring[i] = np.clip(offspring[i], -10, 10)
```

The function np.random.rand generates a random floating point number in the range of 0 and 1 with an equal likelihood of every value in this range. If the number is less than the mutation probability **mc**, the mutation operation is carried out otherwise no mutation operation happens and cloning takes place. Similar to the crossover operation.

The offspring then replaces the population.

```
population = np.array(offspring)
```

Additionally, the best solution from each generation is stored. As a result, after all generations are completed, we have a set of solutions corresponding to each generation, from which the best-fit solution can be selected as the optimal one.

These steps are repeated until the termination condition is met, which in this case is the completion of 100 generations. The algorithm will run for a maximum of 100 generations.

Streamlit User Interface:

The UI is an interactive application designed to solve polynomial equations using a genetic algorithm. It enables users to dynamically input equations by selecting the polynomial's degree (quadratic, cubic, or higher) and specifying the coefficients for each term.

Key Features

- Users can choose the degree of the polynomial from a dropdown menu, making the app versatile for various types of equations.
- Based on the selected degree, the app generates input fields for coefficients, ensuring clarity and simplicity.
- By clicking the "Run Genetic Algorithm" button, the app calculates the root of the polynomial by evolving a population over multiple generations to maximize fitness.
- The fitness progression is displayed using a line chart, allowing users to observe the algorithm's performance over iterations.
- The app highlights the best solution (root) found along with its corresponding fitness value.

Sample Run

Genetic Algorithm Solver for Polynomial Equations

This app uses a genetic algorithm to find the roots of quadratic, cubic, or custom polynomial equations.

Select the degree of the polynomial equation:

4

Enter the coefficients for your degree-4 polynomial:

$$a_4x^4 + a_3x^3 + \ldots + a_1x + a_0 = 0$$

Coefficient for x^4:

1.00

Coefficient for x^3:

-6.00

Coefficient for x^2:

5.00

Coefficient for x^1:

-24.00

Polynomial coefficients (highest to lowest degree):


```
[
  0 : 1
  1 : -6
  2 : 5
  3 : -24
  4 : 26
]
```


Best solution found: 1.0713595294491756

Best fitness value: 0.966731928726513