Factuality of Engineering & Technology

Department of Computer Science

Comp338 - Artificial Intelligence

-Project 1-

Prepared By

Name: Rama Shaheen | **Student ID:** 1211153

Name: Hind Sumary | **Student ID:** 1213151

Instructor's Name: Mohammed Helal

Section: 2

Date: 05/01/2025

Table Of Content

Table Of Content **2**
Genetic Algorithm **3**
Steps of Genetic Algorithm 3
Setting Parameters 3
Problem Formulation **4**
Objective: 4
Explanation of the Code **4**
Code for Initialization 4
Fitness Evaluation 4
Selection 5
Crossover 5
Mutation 5
Main Genetic Algorithm Loop 6
Parameter Tuning Analysis **7**
Convergence Rate Plot 8
Screenshots **9**
Discussion **12**
Key Observations 13
References **13**

Genetic Algorithm

Widely applied to solve optimization and search problems, finding high-quality solutions is critical.

Search techniques that are inspired by the concept of natural selection and genetics, using historical data to guide random searches, help to focus on areas in the solution space that are more likely to return better results.

Steps of Genetic Algorithm

1. **Initialization:** Generate an initial population of random candidate solutions (chromosomes).
2. **Fitness Evaluation:** Assess each chromosome's performance using a fitness function.
3. **Selection:** Select parent chromosomes based on their fitness scores.
4. **Crossover:** Combine parts of two parent chromosomes to produce offspring.
5. **Mutation:** Introduce small random changes to offspring to maintain diversity.
6. **Termination:** Repeat the steps until a stopping criterion is met (e.g., an exact match or maximum generations).

Setting Parameters

- **Population Size:** Number of chromosomes in each generation.
- **Mutation Rate:** Probability of a gene mutating.
- **Crossover Rate:** Probability of two parents undergoing crossover.
- **Stopping Criteria:** Maximum number of generations or reaching the desired solution.

Problem Formulation

Representation

- **Solution Representation:** Each candidate solution is a 32-bit binary sequence.
- **Chromosome:** Represents an entire 32-bit sequence.
- **Gene:** Represents a single binary digit (0 or 1) within the chromosome.

Objective:

Find the exact 32-bit passcode that was randomly generated at the start of the program.

Explanation of the Code

Below, we explain each stage of the implemented genetic algorithm:

Code for Initialization

The initial population is a list of 32-bit binary sequences generated randomly.

```
def initialize_population():
    return [[random.randint(0, 1) for _ in range(LENGTH_PASS)] for _ in
range(POPULATION)]
```

Fitness Evaluation

The fitness function calculates how many bits in a chromosome match the target passcode.

```
def calculate_fitness(chromosome, passcode):
    return sum(1 for gene, target in zip(chromosome, passcode) if gene ==
target)
```

Selection

Parent chromosomes are selected based on their fitness scores using roulette wheel selection.

```
def select_parents(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    selection_probs = [score / total_fitness for score in fitness_scores]
    return random.choices(population, weights=selection_probs, k=2)
```

Crossover

Crossover combines segments of two parent chromosomes to produce offspring.

```
def crossover(parent1, parent2):
    point = random.randint(1, LENGTH_PASS - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2
```

Mutation

Mutation flips bits in the offspring with a small probability.

```
def mutate(chromosome):
    return [gene if random.random() > MUTATION_RATE else 1 - gene for gene in
chromosome]
```

Main Genetic Algorithm Loop

The algorithm evolves the population across generations until the target is found.

```
def genetic_algorithm():

    # call the Generate target passcode method
    passcode = generate_passcode()
    print("Generated Passcode:", ''.join(map(str, passcode)))

    # call the Initialize population method
    population = initialize_population()
    fitness_history = []

    for generation in range(GENERATIONS):
        # Calculate fitness for each chromosome by calling the fitness function
        fitness_scores = [calculate_fitness(chrom, passcode) for chrom in
population]
        fitness_history.append(max(fitness_scores))

        # Check if the passcode was found
        if max(fitness_scores) == LENGTH_PASS:
            print(f"Passcode cracked in generation {generation + 1}")
            print("Best Chromosome:", ''.join(map(str,
population[fitness_scores.index(max(fitness_scores))])))
            break

        # Create new population
        new_population = []
        for _ in range(POPULATION // 2):
            parent1, parent2 = select_parents(population, fitness_scores)
            child1, child2 = crossover(parent1, parent2)
            new_population.extend([mutate(child1), mutate(child2)])

        population = new_population

    # if the passcode was not found print that the operation failed
    else:
        print("Failed to crack passcode within the maximum generations.")

# Run the Genetic Algorithm
genetic_algorithm()
```

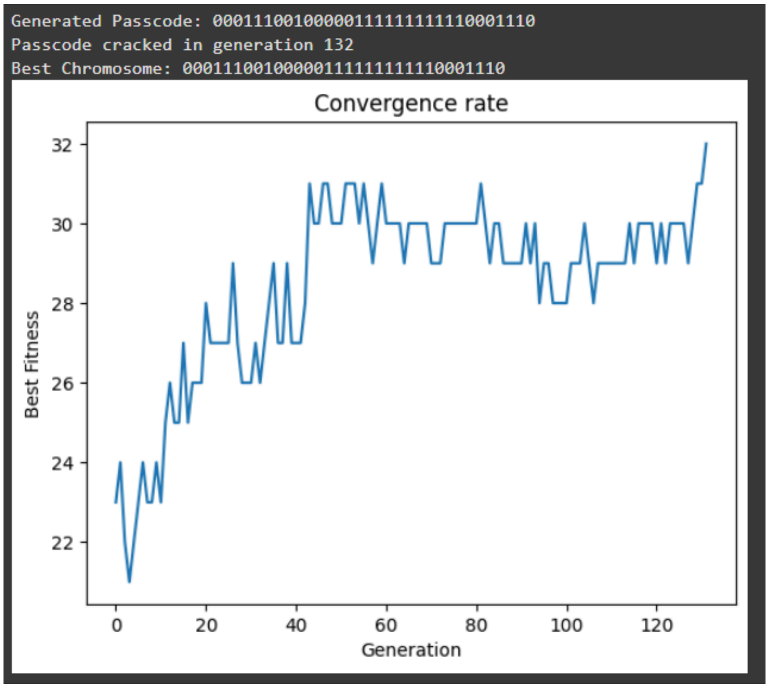
Parameter Tuning Analysis

To evaluate the impact of different parameters on the convergence rate, experiments were conducted with varying population sizes, mutation rates, and stopping criteria. Results are summarized below:

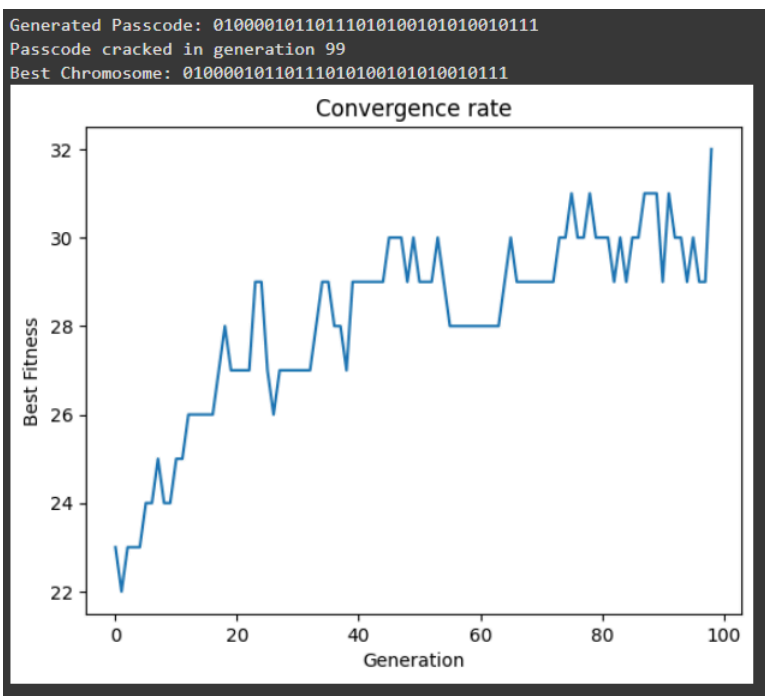
Population Size	Mutation Rate	Generations to Convergence	Execution Time (s)
50	0.01	132	0.422s
50	0.05	1000+ (Failed)	1.043s
50	0.1	1000+ (Failed)	1.069s
100	0.01	99	0.004s
100	0.05	1000+ (Failed)	2.013s
100	0.1	1000+ (Failed)	2.075s
200	0.01	51	0.357s
200	0.05	1000+ (Failed)	4.779s
200	0.1	1000+ (Failed)	4.649s

Convergence Rate Plot

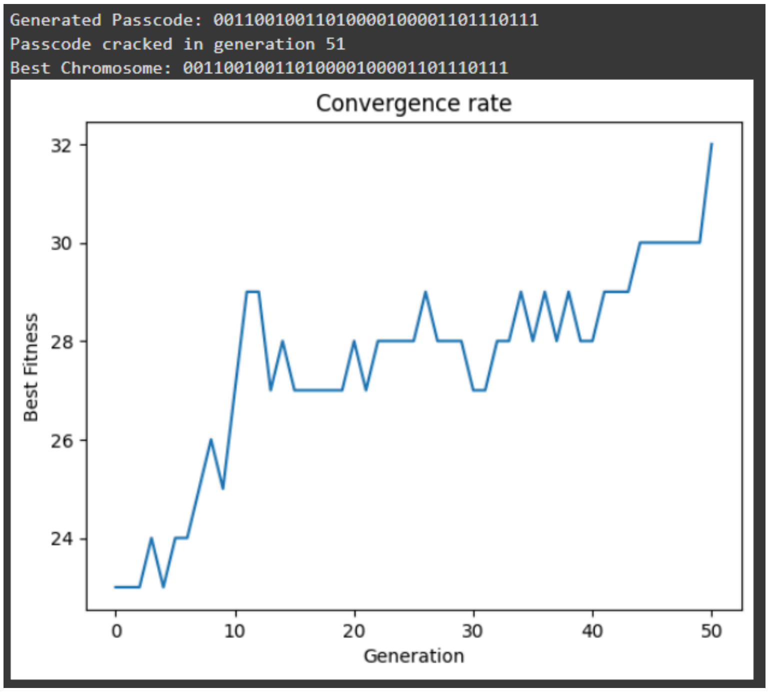
When Population = 50 & mutation = 0.01



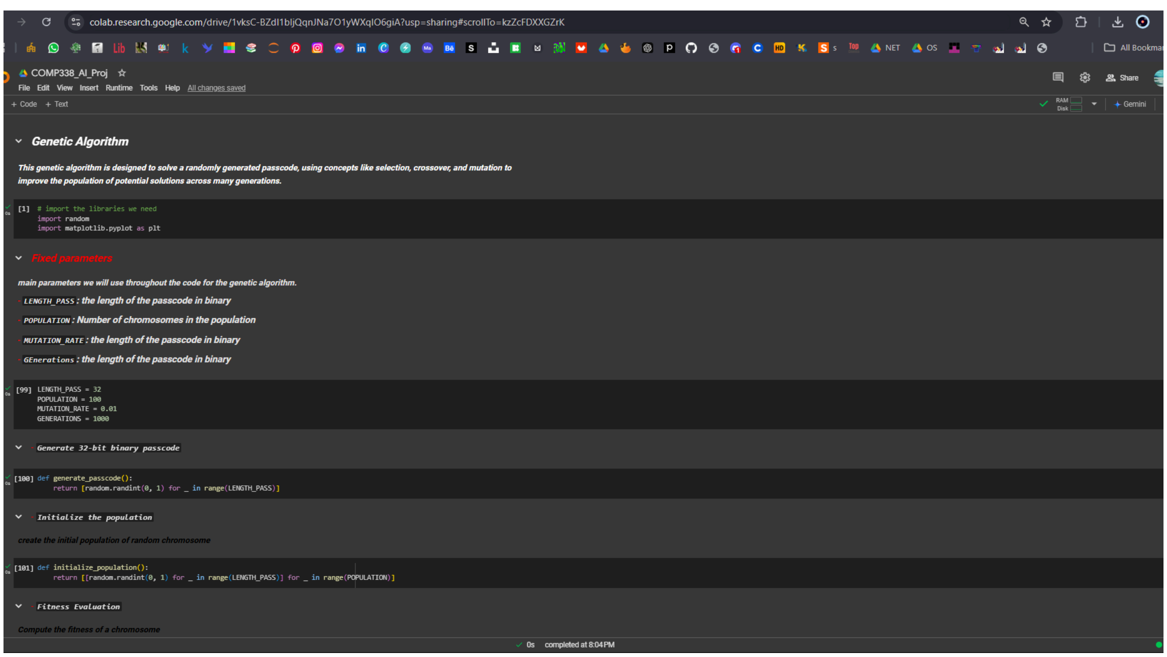
When Population = 100 & mutation = 0.01

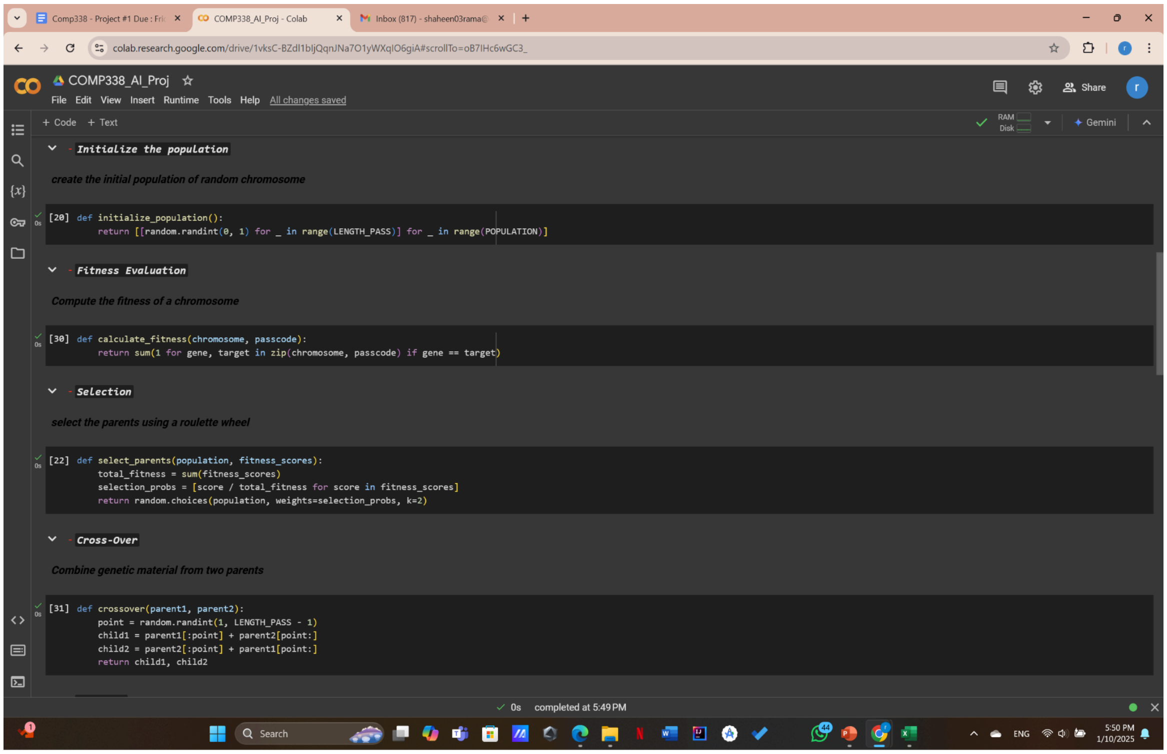


When Population = 200 & mutation = 0.01

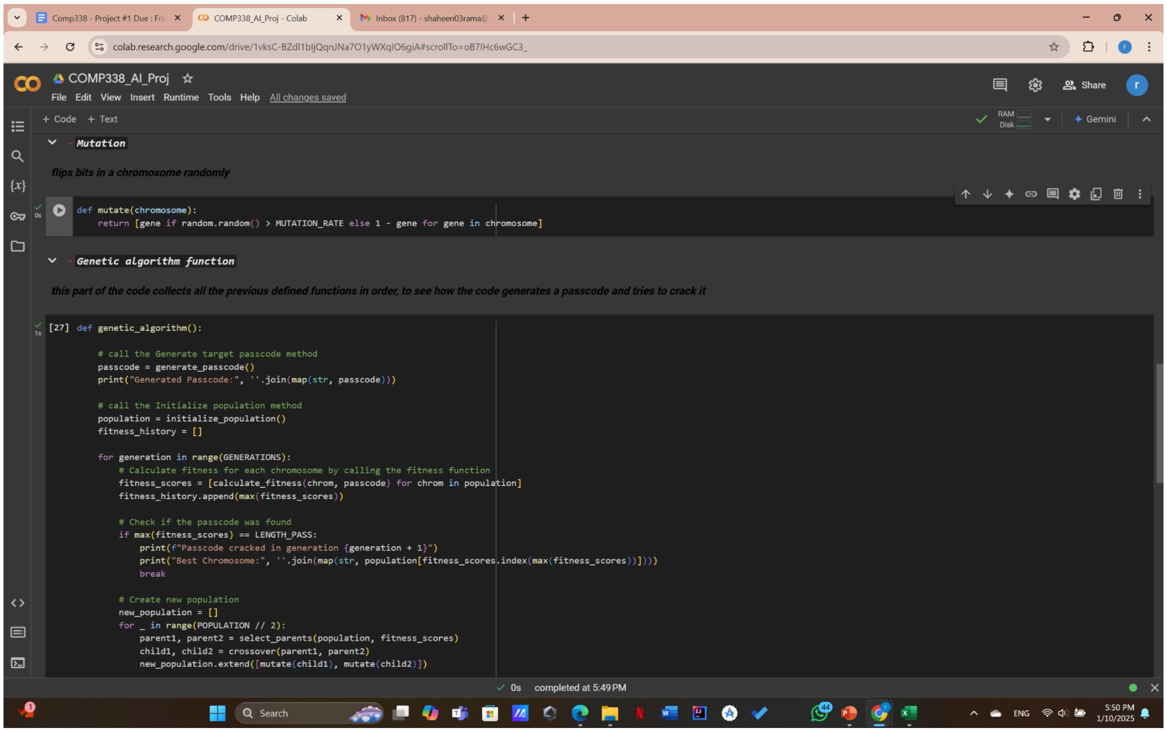


Screenshots

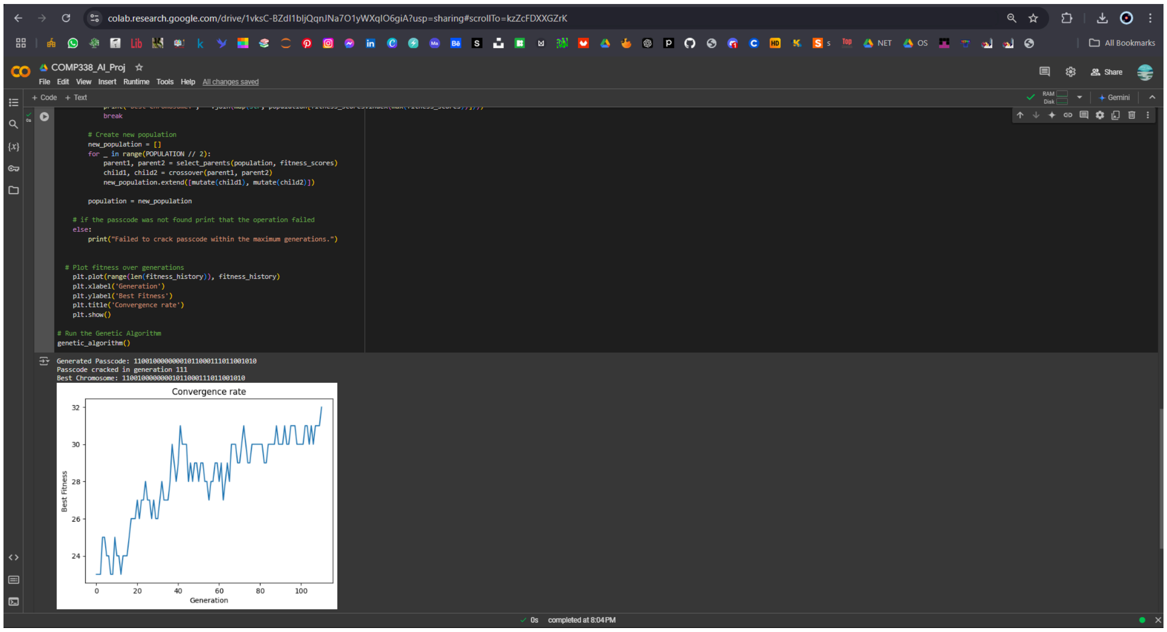




COMP338_AI_Proj
Initialize the population
create the initial population of random chromosome
[20] def initialize_population():
    return [[random.randint(0, 1) for _ in range(LENGTH_PASS)] for _ in range(POPULATION)]
Fitness Evaluation
Compute the fitness of a chromosome
[30] def calculate_fitness(chromosome, passcode):
    return sum(1 for gene, target in zip(chromosome, passcode) if gene == target)
Selection
select the parents using a roulette wheel
[22] def select_parents(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    selection_probs = [score / total_fitness for score in fitness_scores]
    return random.choices(population, weights=selection_probs, k=2)
Cross-Over
Combine genetic material from two parents
[31] def crossover(parent1, parent2):
    point = random.randint(1, LENGTH_PASS - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2
completed at 5:49 PM



COMP338_AI_Proj
Mutation
flips bits in a chromosome randomly
def mutate(chromosome):
    return [gene if random.random() > MUTATION_RATE else 1 - gene for gene in chromosome]
Genetic algorithm function
this part of the code collects all the previous defined functions in order, to see how the code generates a passcode and tries to crack it
[27] def genetic_algorithm():
    # call the Generate target passcode method
    passcode = generate_passcode()
    print("Generated Passcode:", ''.join(map(str, passcode)))
    # call the Initialize population method
    population = initialize_population()
    fitness_history = []
    for generation in range(GENERATIONS):
        # Calculate fitness for each chromosome by calling the fitness function
        fitness_scores = [calculate_fitness(chrom, passcode) for chrom in population]
        fitness_history.append(max(fitness_scores))
        # Check if the passcode was found
        if max(fitness_scores) == LENGTH_PASS:
            print(f"Passcode cracked in generation {generation + 1}")
            print("Best Chromosome:", ''.join(map(str, population[fitness_scores.index(max(fitness_scores))])))
            break
        # Create new population
        new_population = []
        for _ in range(POPULATION // 2):
            parent1, parent2 = select_parents(population, fitness_scores)
            child1, child2 = crossover(parent1, parent2)
            new_population.extend([mutate(child1), mutate(child2)])
completed at 5:49 PM


colab.research.google.com/drive/1vksC-BZdI1bljQqnJNa7O1yWXqIO6giA?usp=sharing#scrollTo=kzZcFDXXGZrK
COMP338_AI_Proj
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
break
# Create new population
new_population = []
for _ in range(POPULATION // 2):
parent1, parent2 = select_parents(population, fitness_scores)
child1, child2 = crossover(parent1, parent2)
new_population.extend([mutate(child1), mutate(child2)])
population = new_population
# if the passcode was not found print that the operation failed
else:
print("Failed to crack passcode within the maximum generations.")
# Plot fitness over generations
plt.plot(range(len(fitness_history)), fitness_history)
plt.xlabel('Generation')
plt.ylabel('Best Fitness')
plt.title('Convergence rate')
plt.show()
# Run the Genetic Algorithm
genetic_algorithm()
Generated Passcode: 11001000000001011000111011001010
Passcode cracked in generation 111
Best Chromosome: 11001000000001011000111011001010
Convergence rate
Best Fitness
Generation
completed at 8:04PM

Discussion

- **Population Size Impact**:
  - Smaller population sizes (**50**) resulted in faster execution times but required significantly more generations to converge due to limited genetic diversity.
  - Larger populations (**200**) converged in fewer generations due to higher diversity, but execution time increased significantly.
- **Mutation Rate Impact**:
  - **Low Mutation Rate (0.01)**:
    - A mutation rate of **0.01** consistently resulted in **successful convergence** across all population sizes.
    - With smaller populations (50), it converged in **132 generations** and had a reasonable execution time of **0.422 seconds**.
    - Larger populations (200) showed the fastest convergence at **51 generations** with an execution time of **0.357 seconds**, indicating that low mutation rates work well with sufficient population diversity.
  - **Moderate Mutation Rate (0.05)**:
    - A mutation rate of **0.05** failed to converge within the maximum number of generations (**1000+ generations**) for all population sizes.
    - This suggests that the moderate mutation rate caused excessive exploration without adequately preserving promising solutions, especially when paired with higher population sizes.
  - **High Mutation Rate (0.1)**:
    - Similarly, a mutation rate of **0.1** failed to converge within **1000 generations** for all population sizes.
    - The high mutation rate introduced too much variability, disrupting the algorithm's ability to refine and converge toward an optimal solution.
- **Best Parameter Combination**:
  - For this problem, a **population size of 100** with a **mutation rate of 0.01** achieved a balance between execution time and convergence. It required only **99 generations** and executed in **0.004 seconds**, offering both efficiency and solution quality.

Key Observations

1. **Mutation Rate 0.01** is the most effective, balancing solution refinement and diversity, with larger populations (e.g., 200) achieving the best performance in terms of convergence speed and execution time.
2. Higher mutation rates (0.05 and 0.1) fail to converge, likely due to excessive disruption of promising solutions during evolution.

References

- Python 3 Documentation
- GeeksforGeeks. (n.d.). *Genetic Algorithms*. Retrieved January 10, 2025, from https://www.geeksforgeeks.org/genetic-algorithms/
- Professor Mustafa jarrar's slides