DEPARTMENT OF COMPUTER SCIENCE

COMP338 - Artificial Intelligence

Course Project 1

”Breaking a Binary Passcode Using a Genetic Algorithm ”

Student Name	Student ID
Habeeb Ali Ahmad	1231086
Ameer Tamimi	1232646

Dr. Mohammad Helal **Section: 1**

`

1. Genetic Algorithm Overview

A Genetic Algorithm (GA) is a search and optimization method inspired by natural evolution. It starts with many possible solutions (a population) and improves them over multiple generations.

GA is useful when the search space is very large and checking every possible solution would take too much time. Instead, the algorithm gradually improves solutions by keeping better candidates and combining them to produce new ones.

The main stages of a GA are: population initialization, fitness evaluation, selection, crossover, mutation, and stopping when a solution is found (or a maximum number of generations is reached).

2. Problem Description

This project breaks a hidden 32-bit binary passcode using a Genetic Algorithm. The program first generates a random 32-bit binary string as the target passcode. The GA does not know the target in advance; it attempts to discover it by evolving a population of candidate bitstrings.

Problem goal: Find the exact 32-bit binary password (target string) using a Genetic Algorithm, instead of brute-forcing **2^32** possibilities.
Search space: All binary strings of length 32 (each position is either 0 or 1).
State/Population: A set of candidate chromosomes in the current generation.
Initial state/first population: Start with a population of completely random 32-bit chromosomes.
Objective function: Maximize the fitness score (higher score means "closer to the target"), and the perfect solution is when the chromosome equals the target exactly.
Goal: Finding the Target/passcode string we generated (an individual matches it)
Stopping criteria: Stop early if the target is found, otherwise stop when the algorithm reaches the maximum number of generations.

3. Chromosome and Gene Representation

In this project, a **gene** represents a single binary value. Each gene can take one of two possible values: 0 or 1.

A chromosome represents a complete candidate solution to the problem. It is composed of 32

genes, where each gene corresponds to one bit in the passcode. Therefore, a chromosome is a 32- bit binary string that represents a guessed version of the passcode.

Each chromosome in the population is evaluated independently, and its quality is determined based on how closely it matches the actual passcode.

4. Fitness Function

The fitness function assigns a numeric score to each chromosome based on how close it is to the target passcode. Higher fitness means a better candidate solution.
First, the chromosome is compared to the target bit by bit. For every position where the bit matches, the fitness score increases.

Next, a constant base value is added to keep fitness values positive. A penalty is also applied if the chromosome has a different number of 1s and 0s than the target, which helps guide the search toward the correct overall structure.

In this project, the fitness score ranges from 0 to 64. A score of 64 means the chromosome matches the target exactly.

```
def calculate_fitness(self):  2 usages (1 dynamic)
    chromosome = self.chromosome
    fitness_value = 0
    for char, gene in zip(self.TARGET, chromosome):
        if char == gene:
            fitness_value += 1

    fitness_value += 32
    fitness_value -= abs(self.ONES - chromosome.count('1'))
    fitness_value -= abs(self.ZEROS - chromosome.count('0'))

    return fitness_value
```

Figure 1: Fitness function.

5. Genetic Algorithm Implementation

The program first generates a random 32-bit target passcode. Then it creates the initial population as random 32-bit chromosomes. Each chromosome is evaluated using the fitness function to measure how close it is to the target.

Selection is then applied so that chromosomes with higher fitness are more likely to be chosen as parents for the next generation.

Crossover combines parts of two parent chromosomes to produce new children. This helps carry useful bit patterns into the next generation.

Mutation flips bits with a small probability. This maintains diversity in the population and helps avoid getting stuck early (premature convergence).

The process repeats generation by generation until an exact match to the target is found (maximum fitness) or the algorithm reaches the maximum number of generations.

6. Results and Convergence

During each run, the program tracks the best fitness value in every generation. These values are saved to a CSV file and later used to draw convergence curves.

A convergence curve shows how fitness improves over generations and how different parameter settings (population size, mutation rate, and crossover rate) affect the speed of reaching the solution.

When the best chromosome reaches the maximum fitness (64), it means the target passcode has been found exactly, and the run stops early.

7. Implementation Details (How the Algorithm Works)

In this section, we explain the logic behind the code we developed to solve the 32-bit passcode problem.

7.1 Fitness Function Logic

Instead of just checking if the bits are identical, we designed a smarter fitness function to guide the algorithm. Our approach follows these steps:

- **Direct Matching:** The program compares each bit in the chromosome with the same bit in the target passcode. For every match, the individual gets a point.
- **Balance Penalty:** We added a special logic that counts the number of 1s and 0s in the chromosome. If the count is different from the target passcode's counts, we subtract points (Penalty).
- **Why we did this?** This helps the algorithm "feel" the correct structure of the passcode even if the bits are not in the exact right positions yet, which leads to faster convergence.

7.2 Crossover Strategy (Producing New Solutions)

To ensure we get the best combination of genes, we used a **Hybrid Crossover** approach:

- **Single-Point Crossover:** With a 60% probability, the program cuts two parents at the midpoint and swaps their halves.
- **Two-Point Crossover:** In other cases, the program cuts the chromosome into three parts and swaps the middle section only.
- **Goal:** This variety prevents the population from becoming "stale" or stuck. It keeps the search dynamic by trying different ways to mix the parents' bits.

```
def cross_over(self, parent2):  1 usage (1 dynamic)
    parent1 = self.chromosome

    single_r = random.random()
    if single_r < self.SINGLE_POINT_RATE:

        # We'll be making a single point cross over (the middle)
        # so ch1 will be having the first part of pa2 and second part of pa1, and ch2 the opposite  way
        mid = self.N // 2
        child1 = parent2[:mid]
        child2 = parent1[:mid]

        # here ch1 will have the second part from pa1, and ch2 from pa2
        child1 += parent1[mid:]
        child2 += parent2[mid:]

    else:
        # Here we will make a 2-point cross over
        # so ch1 will be having the first part of pa1, second from pa2 , third from p1 , and ch2 the opposite way
        slice = self.N // 3
        child1 = parent1[:slice] + parent1[2*slice:]
        child2 = parent1[slice:slice*2]

        child1 += parent2[slice:slice*2]
        child2 += parent2[:slice] + parent2[2*slice:]

    return (child1, child2)
```

Figure 2: Crossover.

`

7.3 Selection Mechanism (The Roulette Wheel)

We used the **Roulette Wheel Selection** method. In this method:

- Every individual has a chance to be a parent, but those with higher fitness scores get a larger "slice" of the wheel.
- This means stronger individuals are more likely to pass their genes, but we also keep some weaker ones to maintain **Genetic Diversity**, which is important to find the hidden passcode.

```
def selection(self, population):  2 usages (2 dynamic)
    total_fitness = 0

    for individual in population:
        total_fitness += individual.fitness

    if total_fitness == 0:  # All individuals are basically trash (no offense tho)
        return population[0]
    r = random.random()
    total_sofar = 0
    for individual in population:
        individual_fitness = (individual.fitness / total_fitness)
        total_sofar += individual_fitness
        if total_sofar >= r:
            return individual
```

Figure 3:Selection

7.4 Mutation (The Random Flip)

Mutation is our way to explore new possibilities. Our code goes through every bit in the chromosome and, based on a very small Mutation Rate, it might flip a 0 to 1 or vice versa.

- This is very important because it prevents the GA from getting stuck in a "local optimum" where it can't find the last few bits of the passcode.

```
def mutation(self, chromosome):  1 usage (1 dynamic)
    for i in range(self.N):
        r = random.random()
        if r < self.MUTATION_RATE:
            chromosome[i] = self.random_gene()

    return chromosome
```

Figure 4: Mutation step.

`

8. Runtime Process Logs

During execution, the program prints a short log for each experiment run. These logs help track whether the passcode was found and how close the best chromosome is to the target.

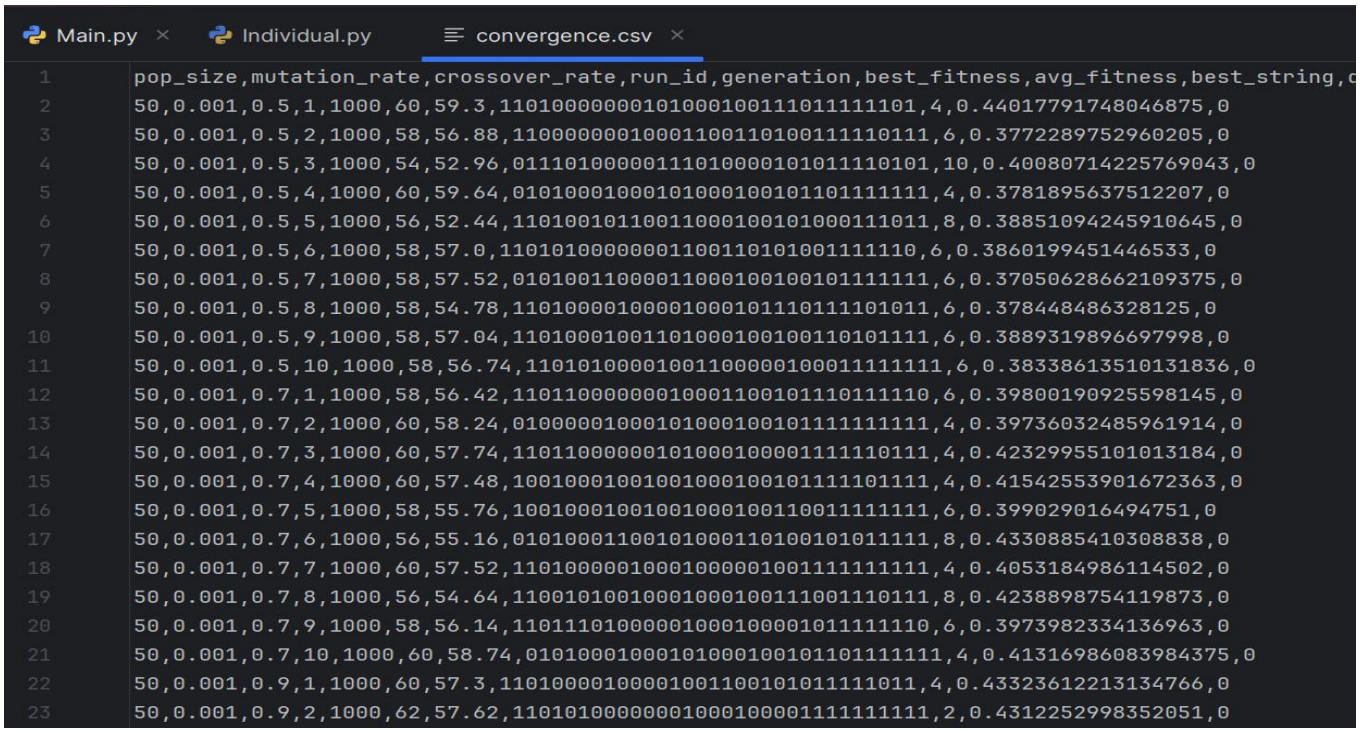
Status labels

Label	Meaning
[FOUND]	The algorithm found an exact match to the 32-bit target.
[NOT FOUND]	The algorithm reached the maximum number of generations without an exact match.

Each log line contains the following fields:

Field	Meaning
pop	Population size (number of chromosomes in each generation).
mut	Mutation rate (probability of flipping a bit).
cross	Crossover rate (probability of applying crossover to parents).
run	Run index for the same configuration.
bestFitness	Fitness score of the best chromosome in that run (maximum = 64).
dist	Hamming distance (number of mismatched bits) from the target.
best	Best chromosome string at that point in the run.

In addition, the program saves per-run and per-generation data to a CSV file (iterations.csv).



`

7.5 Code Walkthrough (Main.py and Individual.py)

The implementation is split into two files. Main.py is responsible for running experiments (different population sizes, mutation rates, and crossover rates), controlling the generation loop, and saving results to a CSV file. Individual.py defines the Individual class, which stores a chromosome and provides the main GA operators (fitness, selection, crossover, and mutation).

The algorithm follows the standard Genetic Algorithm pipeline: initialize a population of random 32-bit chromosomes, evaluate fitness, select parents, apply crossover to create children, apply mutation to maintain diversity, build the next generation using elitism, and repeat until the target is found or the maximum number of generations is reached.

High-level flow of one GA run:

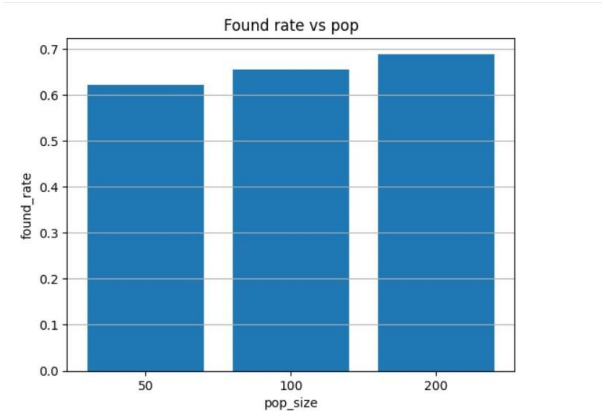
```
1) Generate a random 32-bit TARGET
2) Create POPULATION_SIZE random chromosomes
3) For generation = 1 .. MAX_GEN:
   - Compute fitness for each chromosome
   - Keep top 10% (elitism)
   - Remove worst 30%
   - While new generation is not full:
       * Select parent(s) using roulette-wheel selection
       * Apply crossover with CROSS_OVER_RATE
       * Apply mutation with MUTATION_RATE
   - If best chromosome equals TARGET: stop (FOUND)
4) Save best fitness and distance to CSV for plotting
```

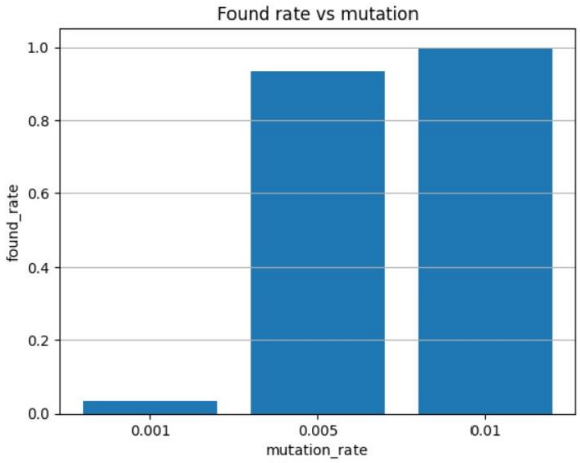
During execution, the program prints a short log line for each run showing the configuration (population size, mutation rate, crossover rate), the best fitness reached, and the remaining distance (number of incorrect bits). The same information is also written to a CSV file to create convergence and parameter tuning plots.

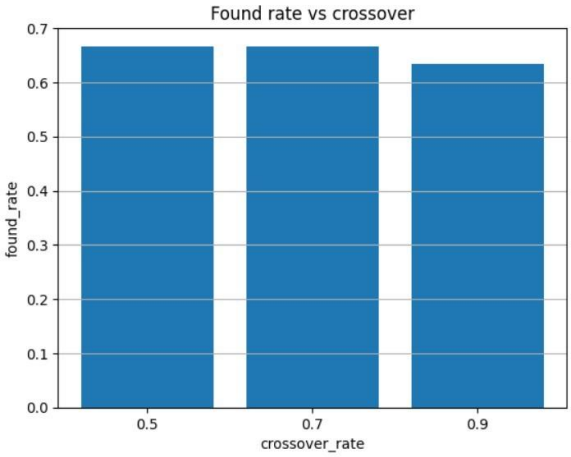
`

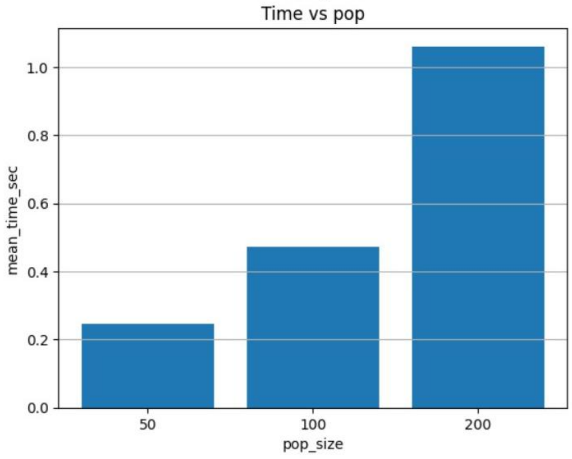
9. Parameter Tuning Plots

The following figures summarize the convergence behavior of the Genetic Algorithm under different parameter settings. The plots are included for comparison only.

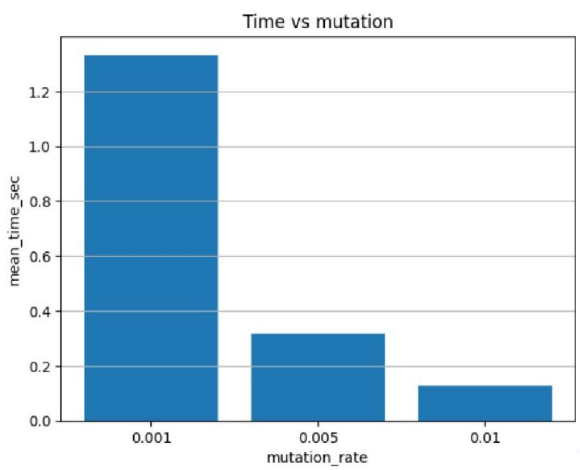


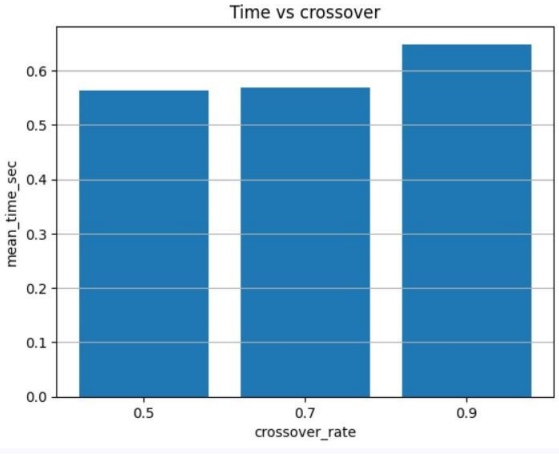


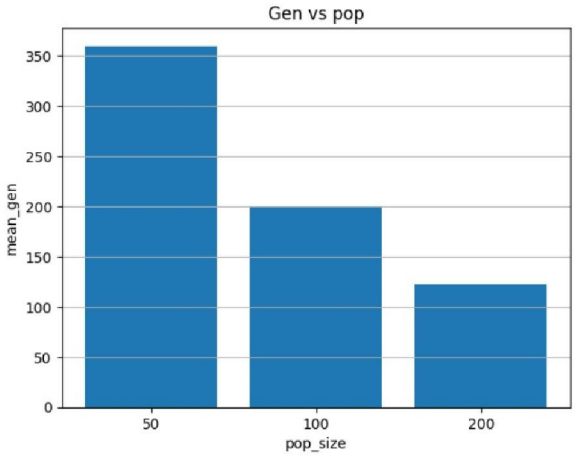


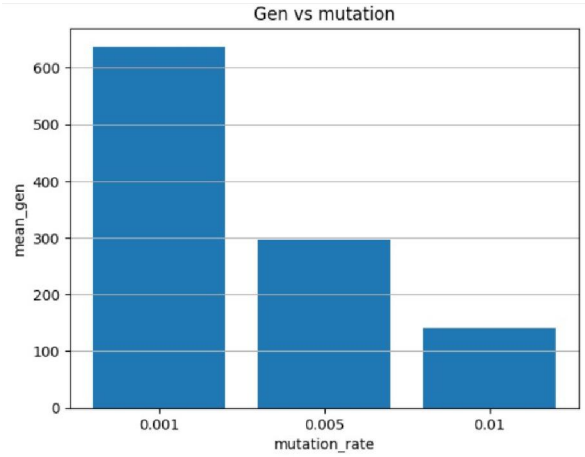


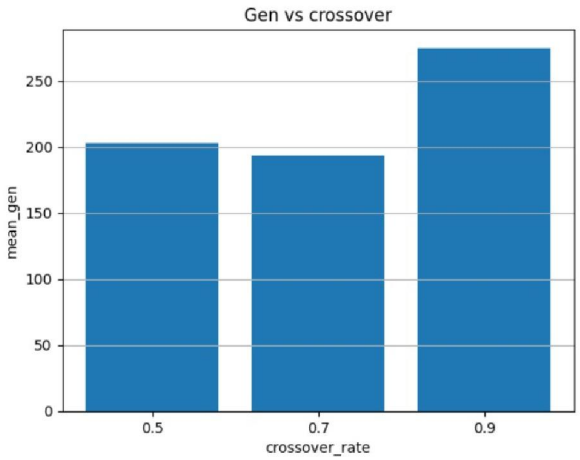
`

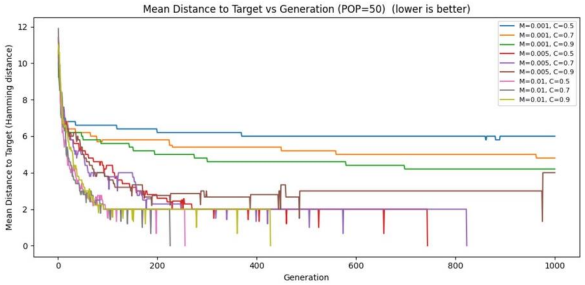

Time vs mutation
mean_time_sec
mutation_rate
0.001
0.005
0.01

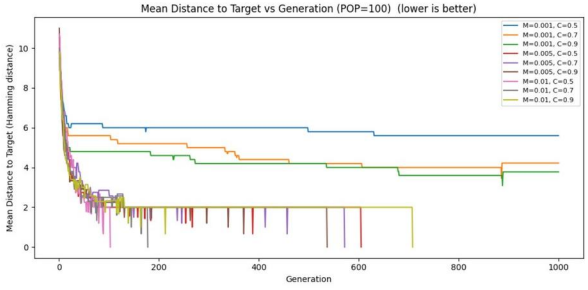

Time vs crossover
mean_time_sec
crossover_rate
0.5
0.7
0.9

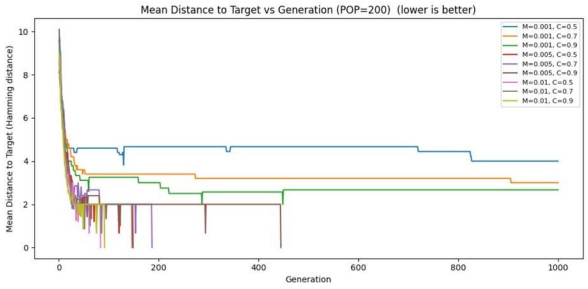

Gen vs pop
mean_gen
pop_size
50
100
200


Gen vs mutation
mean_gen
mutation_rate
0.001
0.005
0.01


Gen vs crossover
mean_gen
crossover_rate
0.5
0.7
0.9


Mean Distance to Target vs Generation (POP=50) (lower is better)
Mean Distance to Target (Hamming distance)
Generation


Mean Distance to Target vs Generation (POP=100) (lower is better)
Mean Distance to Target (Hamming distance)
Generation


Mean Distance to Target vs Generation (POP=200) (lower is better)
Mean Distance to Target (Hamming distance)
Generation

Figures 6-17: Parameter tuning plots.

Explanation *(for time only plots , since the report asked for that explicitly and I actually have literally no time left I wanna sleep so badly god forbid i legit work 24/7 , sorry needed this crashout)*

Population vs Time plot (only when the target is found):

We can see that when the population increases , time needed to find the target actually also increases (make sense , since we need more calculation for each individual)

Mutation vs Time plot : We can clearly see that when mutation increases , the time needed decreases (make sense , since we don't fall into local minima and actually make new changes faster so we find the target faster)

CroosOver vs Time plot: all results almost the same , but when we increase the cross over rate , the slightly goes up , ig because we over do cross over (0.9 is lowkey insane).

10. References

Websites

1. GeeksforGeeks. Genetic Algorithms. https://www.geeksforgeeks.org/dsa/genetic-algorithms/
2. Machine Learning Mastery. Simple Genetic Algorithm From Scratch in Python. https://www.machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/
3. Ultra Evolution Wiki. String Guessing Tutorial. https://ultraevolution.org/wiki/string_guessing_tutorial/

Python Libraries

1. **random**
2. **time**
3. **csv**
4. **pandas**
5. **numpy**
6. **matplotlib.pyplot**