A Genetic Algorithm for Constant Area Coding (CAC) Optimization

CSE489: Selected Topics in Data Science

Supervised By:
Prof. Dr. Alaa Hamdy

Submitted By:
Youssef George Fouad Saad 19P9824

Semester:
Spring 2024

GitHub Repo:
youssefg7/GeneticAlgorithmCAC

Executive Summary

The Genetic Algorithm for Block Optimization is designed to enhance data compression efficiency by optimizing the block width and height parameters within the Constant Area Coding (CAC) framework. Data compression plays a critical role in reducing storage requirements and minimizing data transmission costs. Traditional methods often rely on fixed or heuristically determined block parameters, which may not yield optimal compression ratios. This project leverages the power of genetic algorithms—a nature-inspired optimization technique—to dynamically find the optimal block dimensions that maximize the lossless compression ratio. By simulating the processes of natural selection, crossover, and mutation, the genetic algorithm evolves a population of potential solutions over several generations, converging towards an optimal configuration.

The project encompasses a comprehensive implementation and evaluation of the genetic algorithm, utilizing Python and essential libraries. The methodology involves initializing a diverse population of potential block configurations, evaluating their fitness based on the resulting compression ratio, and iteratively refining the population through selection, crossover, and mutation.

Table of Content

1 Introduction

Foreground detection in video streams is a fundamental task in the field of computer vision, with applications spanning security surveillance, traffic monitoring, interactive media, and more. The challenge lies in accurately distinguishing between static background and moving foreground objects in real-time, under varying lighting conditions and dynamic scene changes. Traditional methods often fall short in complex environments, leading to the need for more adaptive and robust approaches.

1.1 Motivation

In the digital age, the need for efficient data storage and transmission has never been more critical. As the volume of digital information continues to grow exponentially, optimizing data compression techniques has become a priority for researchers and industry professionals alike. Lossless compression methods, which allow the original data to be perfectly reconstructed from the compressed data, are particularly valuable in contexts where data integrity is paramount, such as in medical imaging, scientific data, and archival storage. Traditional compression algorithms often rely on fixed or heuristically chosen parameters that may not fully exploit the potential for optimization. This project aims to address this limitation by employing a genetic algorithm to dynamically optimize block width and block height parameters in Constant Area Coding (CAC), thereby maximizing the compression ratio and improving overall efficiency.

1.2 Objective

The primary objective of this project is to develop a robust and efficient genetic algorithm capable of finding the optimal block dimensions for CAC. This project aims to demonstrate the effectiveness of genetic algorithms in solving complex optimization problems and also to provide a practical tool for improving data compression techniques in real-world applications.

1.3 Scope

This project focuses on the implementation and evaluation of a genetic algorithm for optimizing block parameters in CAC.

- **Development of the Genetic Algorithm:** Designing and coding the algorithm using Python and relevant libraries.
- **Fitness Function Definition:** Establishing a fitness function based on the compression ratio.

- **Performance Evaluation:** Testing the algorithm on various binary pictures and comparing its performance with baseline methods.
- **Documentation and Analysis:** Providing comprehensive documentation of the methodology, results, and findings, along with a detailed analysis of the algorithm's performance.

1.4 Report Structure

This report is organized in sections which are introduction, methodolgy, results, conclusion, and references. This structured approach ensures a comprehensive and systematic exploration of the genetic algorithm's potential in optimizing data compression parameters, providing valuable insights and practical tools for further advancements in the field.

- Section 1: Introduction: This section provides the motivation, objectives, scope, and structure of the report.
- Section 2: Methodology: Details the genetic algorithm's design, implementation, and parameter settings.
- Section 3: Sample Results: Presents the outcomes of the algorithm's application, including performance metrics and comparative analysis.
- Section 4: Conclusion: Summarizes the findings, discusses the implications, and outlines potential directions for future work.

2 Methodology

2.1 Genetic Algorithm

A genetic algorithm (GA) is an evolutionary optimization technique inspired by the principles of natural selection and genetics. It operates on a population of potential solutions, iteratively refining them through processes akin to biological evolution: selection, crossover, and mutation. The goal of the GA in this project is to optimize block width and block height parameters to maximize the lossless compression ratio in Constant Area Coding (CAC).

The key steps involved in a genetic algorithm are:

1. **Initialization:** Generating an initial population of potential solutions.
2. **Fitness Evaluation:** Assessing the fitness of each solution in the population.
3. **Selection:** Choosing the best-performing solutions for reproduction.
4. **Crossover:** Combining pairs of solutions to create new offspring.
5. **Mutation:** Introducing random changes to the offspring to maintain genetic diversity.
6. **Iteration:** Repeating the process over several generations to evolve better solutions.

2.2 Implementation

The genetic algorithm is implemented in Python, utilizing libraries such as NumPy and custom utility functions. The implementation can be broken down into several key components:

2.2.1 Reading and Processing the Image

The first step is to load the image and convert it into a binary format suitable for block processing. This is done using the Python Imaging Library (PIL) and NumPy.

```
images = os.listdir('Dataset') # List all images in the folder
random_image = np.random.choice(images) # Randomly select an image
print('Random Image:', random_image)

img = Image.open('Dataset/' + random_image).convert('1') # Open the image and convert it to binary
img = np.array(img)
```

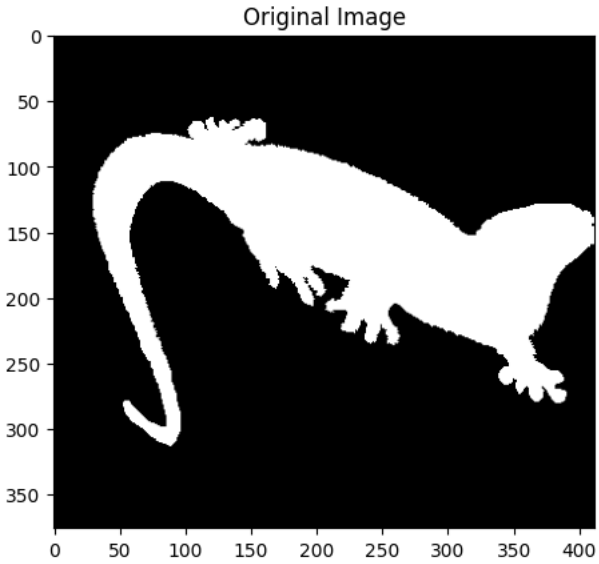


Figure 1: Sample Image

2.2.2 Generating the Initial Population

The initial population consists of chromosomes, each representing a pair of block width and block height. Chromosomes are generated randomly within the specified range of **count of possible block dimensions**.

The possible widths and heights are calculated first using the ***getDivisiors*** method that find all divisors of a number.

```
def getDivisors(n : int) -> list:
    divisors = []
    for i in range(1, n+1):
        if n % i == 0:
            divisors.append(i)
    return divisors
```

```
Image size:  376 x 412
Possible block heights (8):  [1, 2, 4, 8, 47, 94, 188, 376]
Possible block widths (6):  [1, 2, 4, 103, 206, 412]
Possible solutions count (combinations):  48
Population size (10% of the possible solutions to the nearest even number):  4
Chromosome length for block height: 3
Chromosome length for block width: 3
Total chromosome length: 6 bits [ x x x , x x x ]
```

Then, the generate_population method is used to generate 10% of possible solutions initial population according to the chromosome legnth.

```
 def generatePopulation(population_size : int, block_height_chromosome_length : int,
block_width_chromosome_length : int, possible_heights : list, possible_widths : list) -> list:
    population = []
    for _ in range(population_size):
        block_height_index = np.random.randint(0, len(possible_heights))
        block_width_index = np.random.randint(0, len(possible_widths))
        chromosome = np.concatenate((decimalToBinary(block_height_index, block_height_chromosome_length),
decimalToBinary(block_width_index, block_width_chromosome_length)))
        population.append(chromosome)
    return population
```

2.2.3 Fitness Evaluation

The fitness of each chromosome is evaluated based on the compression ratio achieved with the specified block dimensions. The fitness function calculates the ratio of the original image size to the compressed image size.

```
def evaluatePopulation() -> list:
    original_size = original.size
    fitnessValues = []
    for chromosome in population:
        block_height_index = binaryToDecimal(chromosome[:block_height_chromosome_length])
        block_width_index = binaryToDecimal(chromosome[block_height_chromosome_length:])
        if block_height_index >= len(possible_heights) or block_width_index >= len(possible_widths):
            fitnessValues.append(0)
            continue
        block_height = possible_heights[block_height_index]
        block_width = possible_widths[block_width_index]
        compressed_size = CAC(original, block_width, block_height)
        fitness = fitnessFunction(original_size, compressed_size)
        fitnessValues.append(fitness)
    return fitnessValues
```

```
def fitnessFunction(original_size : int, compressed_size : int) -> int:
    return original_size / compressed_size # compression ratio
```

2.2.4 Selection, Crossover, and Mutation

Selection involves choosing the fittest individuals from the population to serve as parents for the next generation. Crossover combines pairs of parent chromosomes to produce offspring, while mutation introduces random changes to the offspring to maintain genetic diversity.

- Select Parents for next generation:

```
def selectSurvivors(population : list, fitness_values : list) -> list:
    survivors_indices = np.argsort(fitness_values)[::-1][:int(len(population) / 2)]
    survivors = [population[i] for i in survivors_indices]
    return survivors
```

- Match parents pairs:

```
def matchParents(survivors : list) -> list:
    available_parents = np.arange(len(survivors))
    parents_tuples = []
    for _ in range(len(survivors)//2):
        parent1_index = np.random.choice(available_parents)
        available_parents = np.delete(available_parents, np.where(available_parents == parent1_index))
        parent2_index = np.random.choice(available_parents)
        available_parents = np.delete(available_parents, np.where(available_parents == parent2_index))
        parents_tuples.append((survivors[parent1_index], survivors[parent2_index]))
    return parents_tuples
```

- Generate Offsprings:

```
def generateOffsprings(parents_tuples, block_height_chromosome_length, block_width_chromosome_length):
    offsprings = []
    for parent1, parent2 in parents_tuples:
        offsprings.append(parent1)
        offsprings.append(parent2)
        children = crossover(parent1, parent2, block_height_chromosome_length,
block_width_chromosome_length)
        children = [mutate(child) for child in children]
        offsprings.extend(children)
    return offsprings
```

- Perform Crossover

```
def crossover(chromosome1 : np.array, chromosome2 : np.array, block_height_chromosome_length : int,
block_width_chromosome_length : int) -> np.array:
    height_crossover_point = np.random.randint(0, block_height_chromosome_length)
    width_crossover_point = np.random.randint(0, block_width_chromosome_length)
    child1 = np.concatenate((chromosome1[:height_crossover_point],
chromosome2[height_crossover_point:block_height_chromosome_length],
chromosome1[block_height_chromosome_length:block_height_chromosome_length+width_crossover_point],
chromosome2[block_height_chromosome_length+width_crossover_point:]))
    child2 = np.concatenate((chromosome2[:height_crossover_point],
chromosome1[height_crossover_point:block_height_chromosome_length],
chromosome2[block_height_chromosome_length:block_height_chromosome_length+width_crossover_point],
chromosome1[block_height_chromosome_length+width_crossover_point:]))
    return [child1, child2]
```

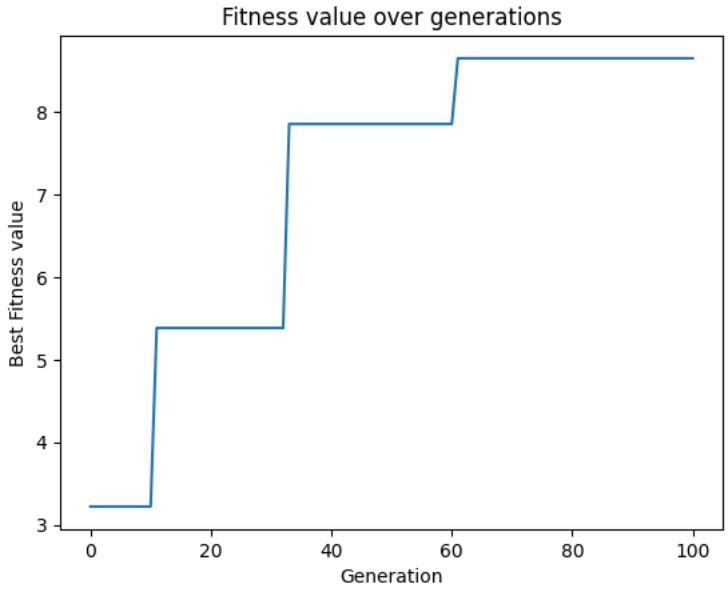
- Perform Mutation:

```
def mutate(chromosome : np.array) -> np.array:
    mutation = np.random.randint(0, 2, chromosome.size)
    mutated_chromosome = np.logical_xor(chromosome, mutation)
    return mutated_chromosome
```

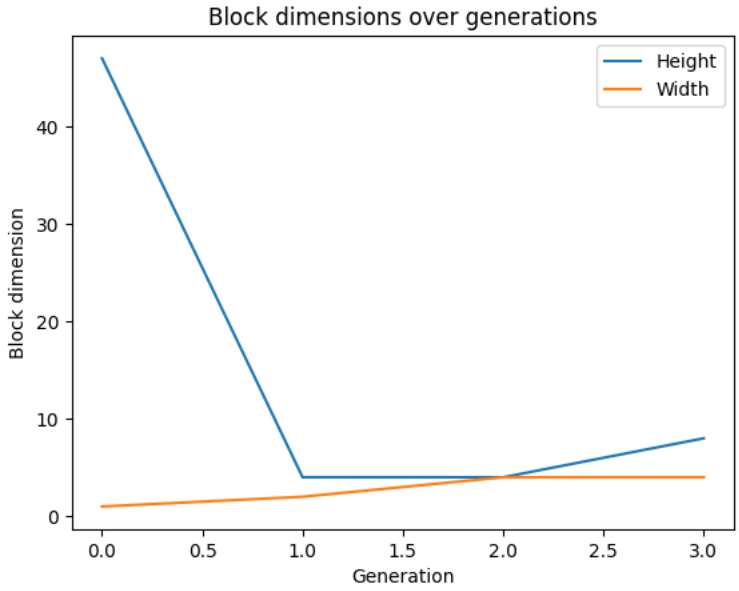
- Iterate:

The genetic algorithm iterates through multiple generations, applying selection, crossover, and mutation at each step. It continues until convergence or satisfactory solution.

```
Generation 0: Best fitness value: 3.221427383130927
Generation 10: Best fitness value: 3.221427383130927
Generation 20: Best fitness value: 5.385808156311929
Generation 30: Best fitness value: 5.385808156311929
Generation 40: Best fitness value: 7.858766233766234
Generation 50: Best fitness value: 7.858766233766234
Generation 60: Best fitness value: 7.858766233766234
Generation 70: Best fitness value: 8.655268745111186
Generation 80: Best fitness value: 8.655268745111186
Generation 90: Best fitness value: 8.655268745111186
Generation 100: Best fitness value: 8.655268745111186
Best fitness value: 8.655268745111186
```




```
Best block size: [8, 4]
```



2.3 Parameters and Settings

Key parameters are tuned through experimentation to achieve the best performance in terms of compression ratio and convergence speed. By following this structured methodology, the genetic algorithm is capable of effectively optimizing block parameters for CAC, leading to significant improvements in data compression efficiency. For the genetic algorithm, they include:

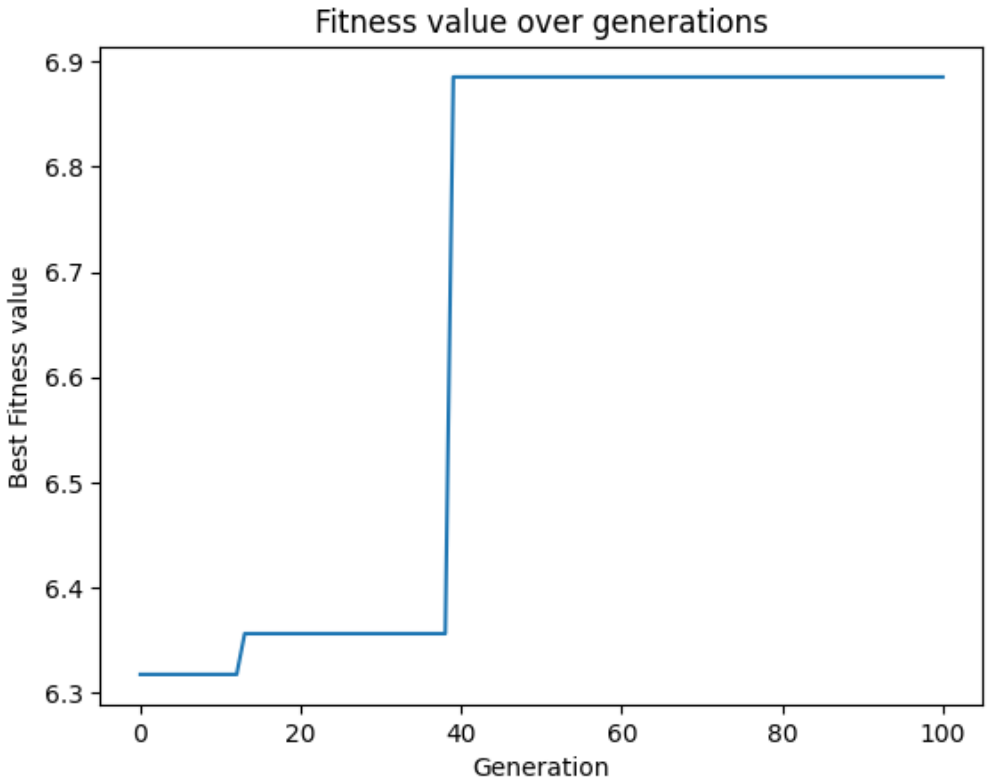
- **Population Size:** The number of chromosomes in the population. A larger population size provides greater diversity but requires more computational resources. We chose to go with **10%** of possible solutions.
- **Mutation Rate:** The probability of introducing random changes to offspring. A higher mutation rate can prevent premature convergence but may also disrupt good solutions.
- **Crossover Rate:** The probability of combining pairs of chromosomes. This parameter determines how often crossover is performed.
- **Number of Generations:** The number of iterations the algorithm runs. More generations allow the algorithm to refine solutions but increase computational time. We chose to perform 100 generations in every run.

3 Sample Results

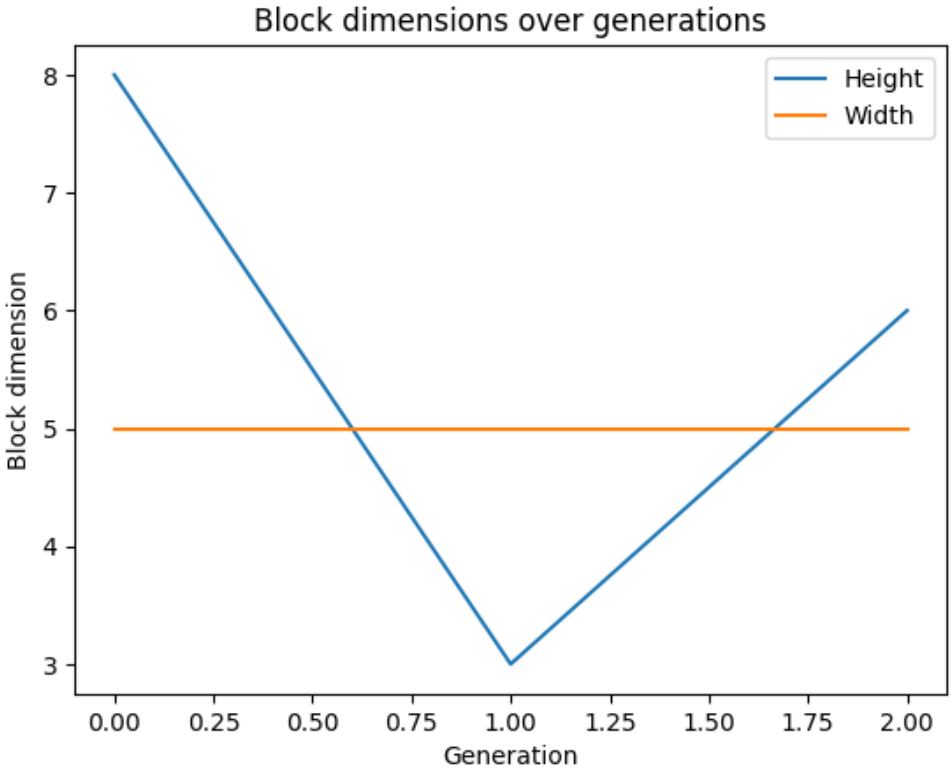
3.1 Sample 1

```
Image size:  168 x 395
Possible block heights (16):  [1, 2, 3, 4, 6, 7, 8, 12, 14, 21, 24, 28, 42, 56, 84, 168]
Possible block widths (4):  [1, 5, 79, 395]
Possible solutions count (combinations):  64
Population size (10% of the possible solutions to the nearest even number):  6
Chromosome length for block height: 4
Chromosome length for block width: 2
Total chromosome length: 6 bits [ x x x x , x x ]
```

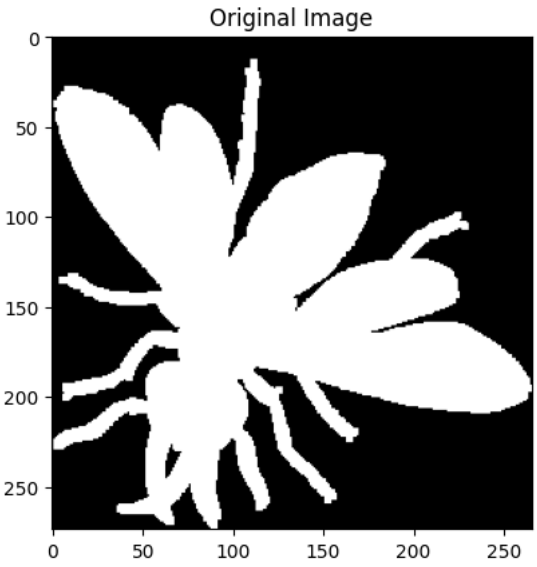
```
Generation 0: Best fitness value: 6.317593297791317
Generation 10: Best fitness value: 6.317593297791317
Generation 20: Best fitness value: 6.35632183908046
Generation 30: Best fitness value: 6.35632183908046
Generation 40: Best fitness value: 6.885245901639344
Generation 50: Best fitness value: 6.885245901639344
Generation 60: Best fitness value: 6.885245901639344
Generation 70: Best fitness value: 6.885245901639344
Generation 80: Best fitness value: 6.885245901639344
Generation 90: Best fitness value: 6.885245901639344
Generation 100: Best fitness value: 6.885245901639344
Best fitness value: 6.885245901639344
```



```
Best block size: [6, 5]
```

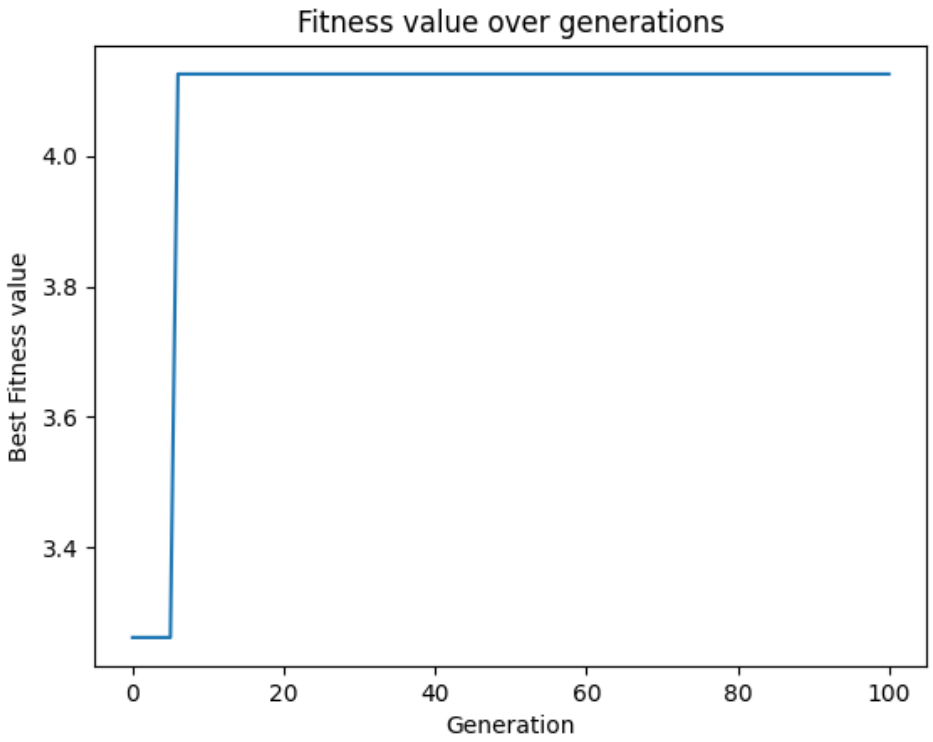


3.2 Sample 2

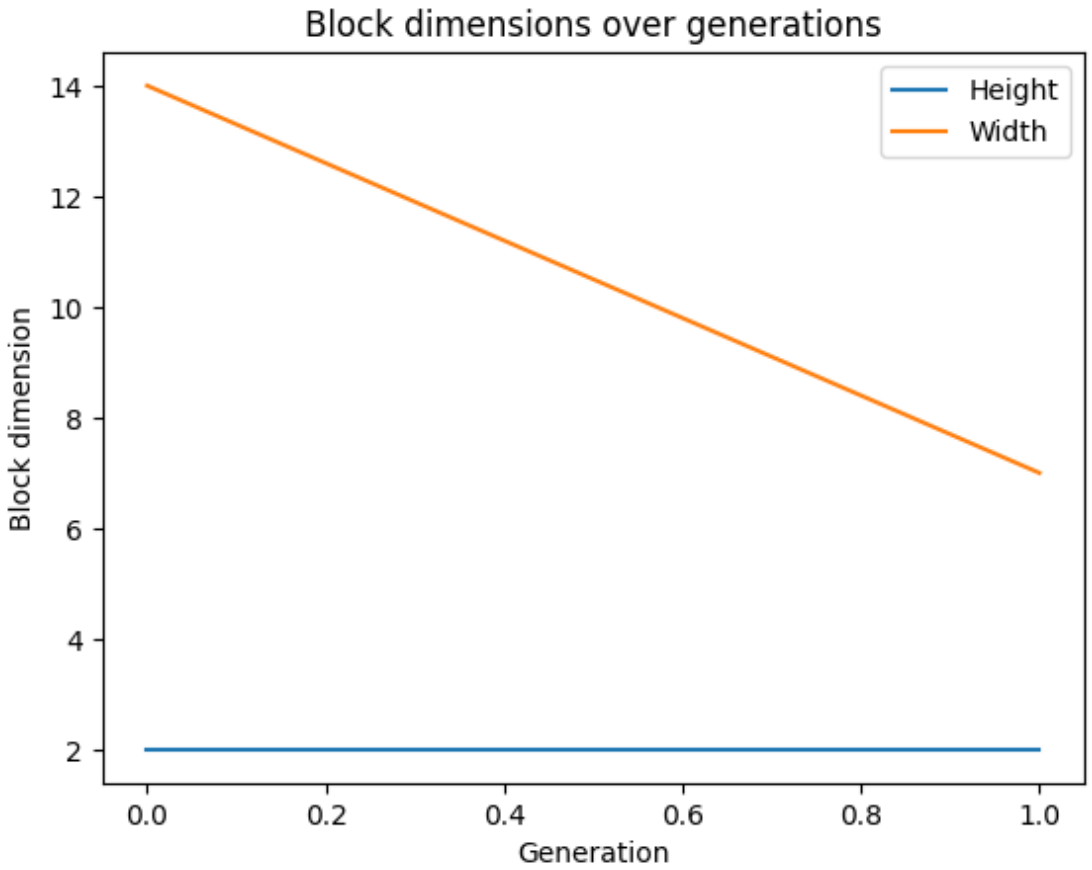


```
Image size:  274 x 266
Possible block heights (4):  [1, 2, 137, 274]
Possible block widths (8):  [1, 2, 7, 14, 19, 38, 133, 266]
Possible solutions count (combinations):  32
Population size (10% of the possible solutions to the nearest even number):  4
Chromosome length for block height: 2
Chromosome length for block width: 3
Total chromosome length: 5 bits [ x x , x x x ]
```

```
Generation 0: Best fitness value: 3.2617587827254417
Generation 10: Best fitness value: 4.125431595630271
Generation 20: Best fitness value: 4.125431595630271
Generation 30: Best fitness value: 4.125431595630271
Generation 40: Best fitness value: 4.125431595630271
Generation 50: Best fitness value: 4.125431595630271
Generation 60: Best fitness value: 4.125431595630271
Generation 70: Best fitness value: 4.125431595630271
Generation 80: Best fitness value: 4.125431595630271
Generation 90: Best fitness value: 4.125431595630271
Generation 100: Best fitness value: 4.125431595630271
Best fitness value: 4.125431595630271
```



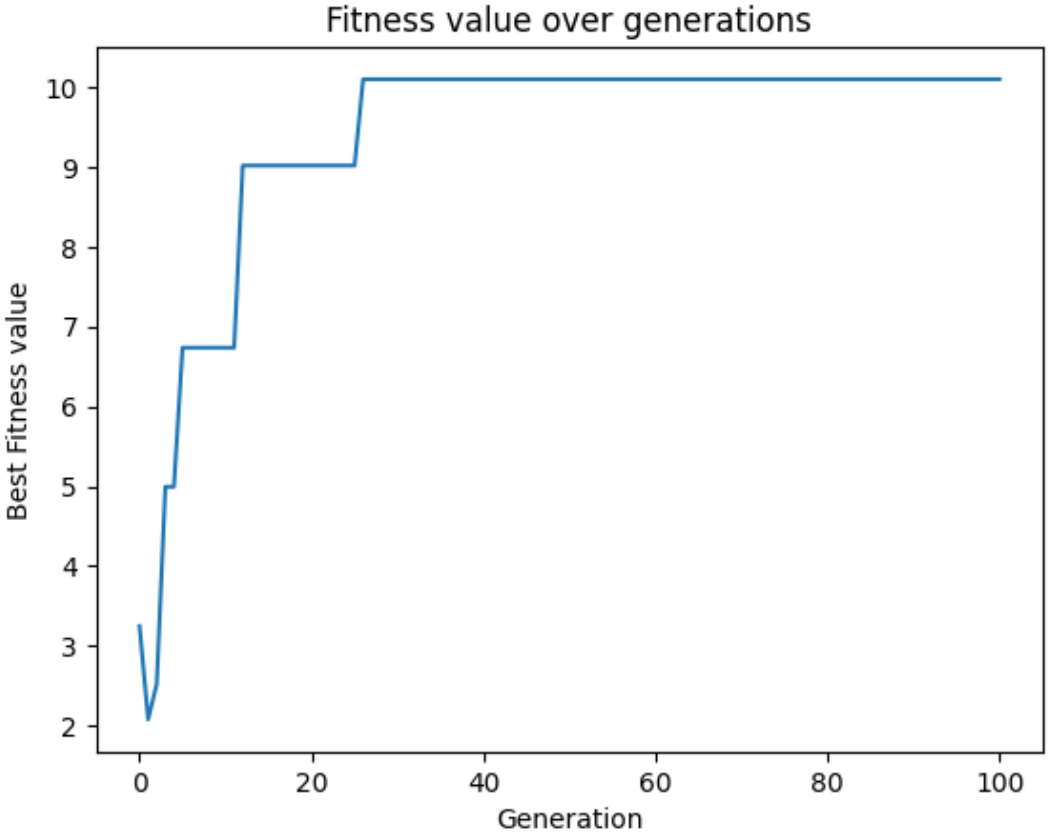
```
Best block size: [2, 7]
```



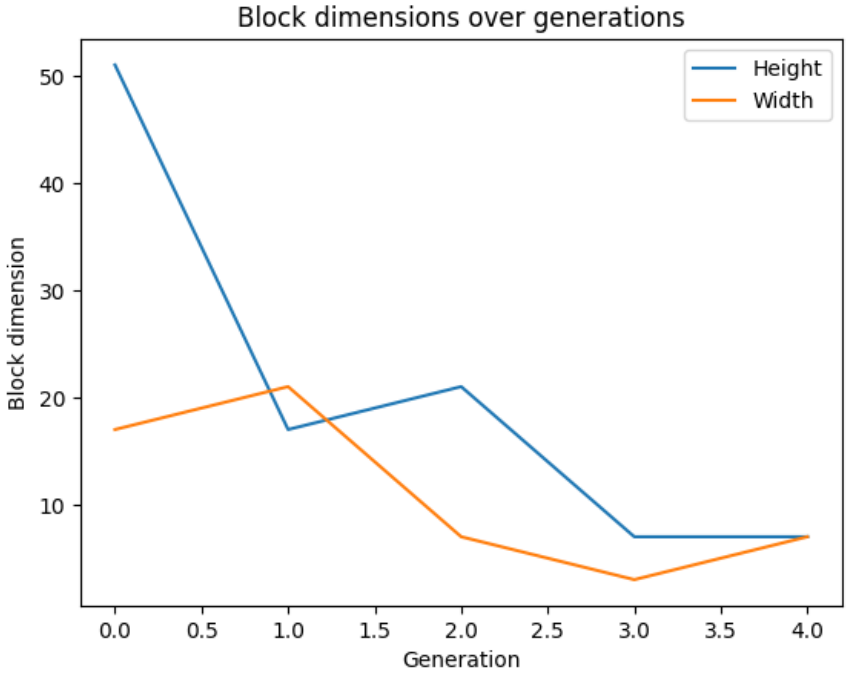
3.3 Sample 3


```
Image size:  357 x 357
Possible block heights (8):  [1, 3, 7, 17, 21, 51, 119, 357]
Possible block widths (8):  [1, 3, 7, 17, 21, 51, 119, 357]
Possible solutions count (combinations):  64
Population size (10% of the possible solutions to the nearest even number):  6
Chromosome length for block height: 3
Chromosome length for block width: 3
Total chromosome length: 6 bits [ x x x , x x x ]
```

```
Generation 0: Best fitness value: 3.2473564858460517
Generation 10: Best fitness value: 6.7351371347038
Generation 20: Best fitness value: 9.016554651574106
Generation 30: Best fitness value: 10.098169717138104
Generation 40: Best fitness value: 10.098169717138104
Generation 50: Best fitness value: 10.098169717138104
Generation 60: Best fitness value: 10.098169717138104
Generation 70: Best fitness value: 10.098169717138104
Generation 80: Best fitness value: 10.098169717138104
Generation 90: Best fitness value: 10.098169717138104
Generation 100: Best fitness value: 10.098169717138104
Best fitness value: 10.098169717138104
```




```
Best block size: [7, 7]
```



4 Conclusion

This project successfully developed and implemented a genetic algorithm to optimize block width and block height parameters for Constant Area Coding (CAC), resulting in significant improvements in the lossless compression ratio. By dynamically adjusting block dimensions, the genetic algorithm was able to achieve a higher compression ratio compared to traditional fixed-parameter methods. The evaluation of the algorithm demonstrated its effectiveness, with a notable increase in compression efficiency and a reasonable execution time. The genetic algorithm converged within 100 generations, indicating a robust and efficient optimization process.

In conclusion, this project has demonstrated the potential of genetic algorithms for optimizing data compression parameters, achieving significant improvements in compression efficiency. The insights gained from this research provide a solid foundation for future advancements in the field, paving the way for more efficient and adaptive data compression techniques.