```python
# ---------------------------------------------------- Imports ----------------------------------------------------
from PIL import Image
import numpy as np
import math
import time
from random import randint, sample, uniform, seed
from itertools import permutations
from datetime import datetime

seed(datetime.now())


# ---------------------------------------------------- Usage Method ----------------------------------------------------
# image_compression(image_reading('Dataset/A2.png'), use_genetic_algorithm=True, debug=False)


# ------------------------------------------------ Image Reading Function ------------------------------------------------
def image_reading(file):
    print("=" * 150)
    print("=" * 68 + " Image Reading " + "=" * 67)
    print("=" * 150)
    print("Image File: " + file)
    image = Image.open(file).convert("1")
    print("Image Mode: " + image.mode)
    (width, height) = image.size
    print("Image Size (Width*Height): (" + str(width) + "*" + str(height) + ")")
    image_array = np.asarray(image)
    print("Binary Image Array:")
    print(image_array)
    return image_array


# -------------------------------------------- Image Compression Function --------------------------------------------
def image_compression(image_array, use_genetic_algorithm=False, debug=False):
    start_time = time.time()
    print("=" * 150)
    print("=" * 51 + " Image Compression: Constant Area Code Algorithm " + "=" * 50)
    print("=" * 150)
    block_widths = divisor_generator(len(image_array[0]))
    block_heights = divisor_generator(len(image_array))
    block_sizes = [(width, height) for width in block_widths for height in block_heights]
    print("# Possible Block Widths:  " + str(block_widths))
    print("# Possible Block Heights: " + str(block_heights))
    print("# Number of Possible Block Sizes (Width*Height): " + str(len(block_sizes)))
    if use_genetic_algorithm:
        max_CR = genetic_algorithm(image_array, block_sizes, debug=debug)
    else:
        max_CR = brute_force(image_array, block_sizes, debug=debug)
    print(" " * 25 + "=" * 100)
    temp_string = str("# Maximum Compression Ratio: " + str(max_CR['CR']))
    print(temp_string)
    print('=' * len(temp_string))
    CAC(image_array, max_CR['block_width'], max_CR['block_height'], get_result=True, debug=True)
    print("=" * 150)
    execution_time = time.time() - start_time
    print("# Program Execution Time: " + str(execution_time) + " Seconds")
    print("=" * 150)


# ------------------------------------------------ Brute Force Function ------------------------------------------------
def brute_force(image_array, block_sizes, debug=False):
    max_CR = {'CR': -1, 'block_width': 0, 'block_height': 0}
    temp_string = str("# Brute Force (Without Optimization):")
    print(temp_string)
    print('=' * len(temp_string))
    if not debug:
        print('Processing', end='', flush=True)
    i = 10
    for block_width, block_height in block_sizes:
        if not debug:
            if i < 150 or i % 150 != 0:
                print('.', end='', flush=True)
            else:
                print('\n.', end='', flush=True)
            i = i + 1
            if (i - 10) == len(block_sizes):
                print()
        else:
            if i == 10:
                i += 1
            else:
                print(" " * 25 + "-" * 100)
        CR = CAC(image_array, block_width, block_height, debug=debug)
        if CR > max_CR['CR']:
            max_CR['CR'] = CR
            max_CR['block_width'] = block_width
            max_CR['block_height'] = block_height
    return max_CR


```

```python
# --------------------------------------------------- CAC Function ---------------------------------------------------
def CAC(image_array, block_width, block_height, debug=False, get_result=False):
    image_width = len(image_array[0])
    image_height = len(image_array)
    blocks_array = np.asarray(
        [[get_block_type(image_array, block_width, block_height, x * block_width, y * block_height)
          for x in range(int(image_width / block_width))] for y in range(int(image_height / block_height))])
    counter, codes = blocks_counter_encoder(blocks_array)
    N1 = image_width * image_height
    N2 = 0
    for key, value in counter.items():
        if key == 'M':
            N2 = N2 + value * (len(codes[key]) + block_width * block_height)
        else:
            N2 = N2 + (value * len(codes[key]))
    CR = N1 / N2
    if get_result:
        with open("Result.txt", "w") as result:
            for h in range(len(blocks_array)):
                for w in range(len(blocks_array[0])):
                    if blocks_array[h][w] == 'M':
                        result.write(codes['M'])
                        for hx in range(h * block_height, (h + 1) * block_height):
                            for wx in range(w * block_width, (w + 1) * block_width):
                                result.write(str(int(image_array[hx][wx])))
                    else:
                        result.write(codes[blocks_array[h][w]])
            result.close()
    else:
        open("Result.txt", "w").close()
    if debug:
        print("For Block Size (Width*Height): (" + str(block_width) + "*" + str(block_height) + ")")
        print("Blocks Counter: " + str(counter))
        print("Blocks Codes:   " + str(codes))
        print("Blocks Array Size (Width*Height): (" + str(len(blocks_array[0])) + "*" + str(len(blocks_array)) + ")")
        print("Blocks Array:")
        print(blocks_array)
        print("Compression Ratio (N1/N2): (" + str(N1) + "/" + str(N2) + ") = " + str(CR))
        print("Result: Result.txt")
    return CR


# -------------------------------------------------- Help Functions --------------------------------------------------
def divisor_generator(n):
    divisors = []
    large_divisors = []
    for i in range(1, int(math.sqrt(n) + 1)):
        if n % i == 0:
            divisors.append(i)
            if i * i != n:
                large_divisors.append(int(n / i))
    return sorted(divisors + large_divisors)


def get_block_type(image_array, block_width, block_height, w_start, h_start):
    has_white = False
    has_black = False
    for w in range(w_start, w_start + block_width):
        for h in range(h_start, h_start + block_height):
            if image_array[h][w]:
                has_white = True
            else:
                has_black = True
            if has_white and has_black:
                return 'M'
    if has_white:
        return 'W'
    elif has_black:
        return 'B'


def blocks_counter_encoder(blocks_array):
    unique, counts = np.unique(blocks_array, return_counts=True)
    counter = dict(zip(unique, counts))
    codes = counter.copy()
    max_count = max(counter, key=counter.get)
    codes[max_count] = '0'
    i = 0
    for key, value in counter.items():
        if key is not max_count:
            if len(counter) == 2:
                codes[key] = '1'
            elif len(counter) == 3:
                if i == 0:
                    codes[key] = '01'
                    i = i + 1
                elif i == 1:
                    codes[key] = '11'
    return counter, codes

```

```python

# -------------------------------------------- Genetic Algorithm Function --------------------------------------------
def genetic_algorithm(image_array, block_sizes, delta_error=10 ** -5, least_number_of_generations=5, debug=False):
    max_CR = {'CR': -1, 'block_width': 0, 'block_height': 0}
    temp_string = str("# Genetic Algorithm (Optimization):")
    print(temp_string)
    print('=' * len(temp_string))
    multiplicat = uniform((3 / len(block_sizes)), 0.1)
    if len(block_sizes) < 30:
        multiplicat = (3 / len(block_sizes))
    if multiplicat <= 1:
        population_size = randint(3, int(len(block_sizes) * multiplicat))
    else:
        population_size = len(block_sizes)
    least_number_of_mating_pools = population_size
    for X in range(population_size):
        if population_size <= (X + len(list(permutations([None] * X, 2)))):
            least_number_of_mating_pools = X
            break
    number_of_mating_pools = randint(least_number_of_mating_pools, population_size)
    print("Least Number of Generations: " + str(least_number_of_generations))
    print("Δ Error of Convergence: " + str(delta_error))
    print("Population Size [3->" + str(int(len(block_sizes) * multiplicat)) + "]: " + str(population_size))
    print("Number of Mating Pools [" + str(least_number_of_mating_pools) + "->" + str(population_size) + "]: " +
          str(number_of_mating_pools))
    print("Mutation Percentage {0%,50%,100%}: 50%")
    if not debug:
        print('Processing', end='', flush=True)
    i = 10
    generations_counter = 0
    new_populations = []
    random_indexes = sample(range(len(block_sizes)), population_size)
    for i in range(population_size):
        new_populations += [block_sizes[random_indexes[i]]]
    condition = True
    while condition:
        if not debug:
            if i < 150 or i % 150 != 0:
                print('*', end='', flush=True)
            else:
                print('\n*', end='', flush=True)
            i = i + 1
        else:
            if i == 10:
                i += 1
            else:
                print(" " * 25 + "-" * 100)
        last_max_CR = max_CR.copy()
        generations_counter += 1
        populations = new_populations.copy()
        populations_with_fitnesses = [(CAC(image_array, width, height), width, height) for width, height in populations]
        populations_with_fitnesses.sort(reverse=True)
        max_CR['CR'] = populations_with_fitnesses[0][0]
        max_CR['block_width'] = populations_with_fitnesses[0][1]
        max_CR['block_height'] = populations_with_fitnesses[0][2]
        mating_pools = []
        for i in range(number_of_mating_pools):
            mating_pools += [(populations_with_fitnesses[i][1], populations_with_fitnesses[i][2])]
        possible_offsprings = [(X1, Y2) for (X1, Y1), (X2, Y2) in list(permutations(mating_pools, 2))]
        offsprings = []
        random_indexes = sample(range(len(possible_offsprings)), population_size - number_of_mating_pools)
        for i in range(population_size - number_of_mating_pools):
            offsprings += [possible_offsprings[random_indexes[i]]]
        mutants = offsprings.copy()
        for i in range(len(mutants)):
            index = randint(0, 1)
            temp_mutant = list(mutants[i])
            temp_mutant[index] = block_sizes[randint(0, len(block_sizes) - 1)][index]
            mutants[i] = tuple(temp_mutant)

        new_populations = mating_pools + mutants
        condition = (generations_counter < least_number_of_generations) or (
                (max_CR['CR'] - last_max_CR['CR']) > delta_error)
        if debug:
            temp_string = "Generation: (" + str(generations_counter) + ")"
            print(temp_string)
            print('-' * len(temp_string))
            print("Populations (BlockWidth,BlockHeight):   " + str(populations))
            print("Sorted Populations with Fitnesses (CR,BlockWidth,BlockHeight): " + str(populations_with_fitnesses))
            print("Mating Pools (BlockWidth,BlockHeight):  " + str(mating_pools))
            print("Offsprings (BlockWidth,BlockHeight):    " + str(offsprings))
            print("Mutants (BlockWidth,BlockHeight):       " + str(mutants))
            print("Best Compression Ratio: " + str(max_CR['CR']))
    if (not debug) and (i < 150 or i % 150 != 0):
        print()
    return max_CR


if __name__ == "__main__":
    image_compression(image_reading('Dataset/A2.png'), use_genetic_algorithm=True, debug=False)
```