CP468 AI – TERM PROJECT

DECEMBER 01, 2019

MAX NIEBERGALL 160623100

SRIRAM VASUTHEVAN 170408710

JUSTIN HARROTT 161449800

RASHA NASRI 164161160

How to Run
To run this program, you must have python 3.7 or higher installed. https://www.python.org/downloads/

Next, install matplotlib using pip. https://pip.pypa.io/en/stable/quickstart/

Then, you can run either dejong_test.py, himmelblau_test.py, or rosenbrock_test.py to see the test's reults display.
This calls the SGA funtion in simple_genetic_algorithm.py and performs mating, reproduction and mutation operations
on as many generations of a population of specified sizes. The program will plot each generation's fittest member's fitness measure.
This graph can be saved as a .png file.

#Note
It is possible to compare 2 different bit mutation methods by alternating the commented out sections in the attempt_mutation() function.

Design Decisions
The only data structure which we used were Python Lists. Lists are dynamic arrays that take O(n) time to initialize, O(1) time to update values and
O(1) amortized time to append to the end of the list (only used for the plot).

As all lists are of fixed size, we could have used Numpy Arrays but we wanted to use vanilla Python as much as possible. The entire script is
written in vanilla Python except for plotting.

We used Python modules to make the code easier to follow. Each objective function is in its own module, along with constants that the SGA function
uses. Functions.py contains several helper functions that are not directly part of the SGA. The SGA function itself calls a different function for
each main step: Reproduction, crossover and mutation.

Benchmark Test Results

Comparing 2 methods of character mutation

Method 1
Chooses one bit randomly if random chance falls into the probability that the string should mutate, if random.random() < probability_of_mutation.

Method 2
Bit by bit, perform mutation if random chance falls into the probability that the bit should mutate, if random.random() < probability_of_mutation.

All graphs are plotted as Fitness Measures (Y-axis [0, 1]) against Generations (X-axis [0, 1000]).

Several tests have shown that method 1 is capable of occasionally outperforming method 2, method 2 is far more consistent and is therefore more reliable.

```
#simple_genetic_algorithm.py

from typing import List
import random
from functions import general_decoder
import matplotlib.pyplot as plt


LENGTH_OF_DECIMAL=4

def initialize(pop_size: int, alnum_set: List[str], var_string_length) -> List[str]:
    """
    Generates pop_size of random strings with characters of their alphanumeric character set.

        Args:
            pop_size (int): Number of strings to be generated
            alnum_set (List[str]): Valid characters to genrate random string from
            var_string_length (int): Number of characters expected to be in each string
        Returns:
            (List[str]): List of pop_size random strings made with characters of their alphanumeric
                         character set
        """

        return ["".join(random.choices(alnum_set, k=var_string_length)) for _ in range(pop_size)]


def perform_reproduction(population, inverse_fitness_func) -> List:
        """
        Takes in a population of strings and probabalistically candidates for mating based on the
        relative fitness of each string (i.e. the more fit members of the population will be picked more often
        and will therefore make up a bigger portion of the mating pool).
        Args:
            population (List[str]): List of strings that make up the mating pool
            inverse_fitness_func (<function>): Benchmark function with known global minima; returned values
                                                closer to zero mean the given arguments are
```

```
                                        closer to optimum
                                Args:
                                    (str): Alphanumeric string
                                    representing number system
                                    value(s)
                                Returns:
                                    (float): floating point
                                    value between 0 to 1;
                                    results closer to zero
                                            are closer to
                                            optimization
        Returns:
        (List[str]): List of strings that make up the new mating
pool generation
        """

        #compile a list of floating point values, numbers closer to 0 have
a higher fitness
        inverse_fitnesses = [inverse_fitness_func(s) for s in population]
        min_inv_fit = min(inverse_fitnesses)
        max_inv_fit =
max(inverse_fitnesses)
        #compile a list of inverse fitness values that are normalized
        to [0,1] with regard to
        #the range of the population's fitness measures, numbers closer
        to 0 have a higher fitness
        inverse_fitnesses = [(X-min_inv_fit)/(max_inv_fit-min_inv_fit+1)
        for X in
inverse_fitnesses]                #adding 1 to avoid division by zero

        #invert list of normalized inverse fitness values so that
        numbers closer to 1 have a higher fitness
```

```
    fitnesses = [1 / ((s)+1) for s in inverse_fitnesses]

    #sum up all found fitness measures
    total_fitness = sum(fitnesses)

    #compile a list of each fitness measure's proportion to the sum of all found fitness measures
    #of the mating pool; to be used as a probability that the condidate will mate
    probabilities_of_reproduction = [fit / total_fitness for fit in fitnesses]

    #create list of randomly chosen strings that are weight biased
    return random.choices(population=population, weights=probabilities_of_reproduction, k=len(population))


def perform_mating(population):
    """
    Takes a list of strings, returns a list of strings after potential mating operations.

    Args:
        population (List[str]): List of strings that make up the mating pool
    Returns:
        (List[str]): List of strings that make up the now potentially modified mating pool
    """

    new_pop = []

    while (len(population) > 0):
        s1 = population.pop(random.randint(0, len(population) - 1))
        s2 = population.pop(random.randint(0, len(population) - 1))

        s1p, s2p = crossover_pair(s1, s2)
        new_pop.append(s1p)
        new_pop.append(s2p)

    return new_pop


def crossover_pair(s1, s2):
    """
    Takes two strings for crossover, randomly determines a crossover point
    between 1 and len(s1)-1 then performs crossover of character data at crossover point.
    Assumes len(s1) = len(s2)

    Args:
        s1 (str): string for mating crossover
        s2 (str): string for mating crossover
    Returns:
        s1p (str): string from mating crossover
    s2p (str): string from mating crossover
    """
    crossover_point = random.randint(1, len(s1) - 2)
    s1p = s1[:crossover_point] + s2[crossover_point:]
    s2p = s2[:crossover_point] + s1[crossover_point:]

    return (s1p, s2p)
```

```
def perform_mutations(population, probability_of_mutation, alnum_set):
    """
Takes a list of strings, returns a list of strings after potential mutation
operation.

    Args:
        population (List[str]): List of strings that make up the mating pool
        probability_of_mutation (float): decimal number between 0 and 1 that represents
```

```
                                                    the liklihood
                                                    that a character will mutate into another character from
                                                    its alphanumeric character set
        alnum_set (List[str]): Valid alphanumeric characters to genrate number-system value-strings from
    Returns:
    (List[str]): List of strings that make up a potentially mutated mating pool
    """

    return [attempt_mutation(s, probability_of_mutation, alnum_set) for s in population]


def attempt_mutation(s, probability_of_mutation, alnum_set):
    """
    Takes a string and probabilisticly performs character mutation

    Args:
        s (str): string to mutate
        probability_of_mutation (float): decimal number between 0 and 1 that represents the liklihood
                                                    that a character will mutate into another character from
                                                    its alphanumeric character set
        alnum_set (List[str]): Valid alphanumeric characters to genrate number-system value-strings from
    Returns:
        None
    """

    #bit flipping
    #on average, method 2 seems to outperform method 1

    # #method 1
    # #choose one bit randomly if random chance falls into the probability  that the string should mutate
    # if random.random() < probability_of_mutation:
    #     bit_to_flip = random.randint(0, len(s) - 1)
    #     s = list(s)
    #     s[bit_to_flip] = random.choice(alnum_set)
    #     return "".join(s)
    # else:
    #     return s

    #method 2
    #bit by bit, perform mutation if random chance falls into the probability  that the bit should mutate
    for i in range(len(s)):
        if random.random() < probability_of_mutation:
            s = s[:i] + str(alnum_set[random.randint(0, len(alnum_set)-1)]) + s[i + 1:]
    return s

# program starts here:
# ------------------------------------------------------------------------------------------------------------------------------------------

def SGA(test_function, pop_size, alnum_set, var_string_length, variable_length,
```

```
domain_min, domain_max,
    number_of_generations, probability_of_mutation):
    """
    Simple Genetic Algorithm that finds global minima of test functions through
    generational mating,
    reproduction and bit mutations while printing out each generation's performance
    results.

```

```
    Args:
        test_function (<function>): Benchmark function that has known optimum values (global min)
        pop_size (int): the number of strings to create for population
        alnum_set (List[str]): Valid alphanumeric characters to genrate number system value-strings from
        var_string_length (int): character length of string that contains one or more number system value-string string variables
        variable_length (int): character length of one number system value-string variable
        domain_min (Union[float,int]): min value of operational domain
        domain_max (Union[float,int]): max value of operational domain
        number_of_generations (int): number of times to mate / mutate the poulation in the attempt to hone in
                                    on the optimal input values for the given test_function
        probability_of_mutation (float): decimal number between 0 and 1 that represents the liklihood
                                        that a character will mutate into another character from
                                        its alphanumeric character set
    Prints:
        table: generational performances, avoiding repeat max performance levels between contiguous generations
    """

    #initialize a random population of pop_size values to be the starting point for optimization attempt
    #returns string with (var_string_length * pop_size) number of characters
    population = initialize(pop_size, alnum_set, var_string_length)

    #small anonymous function used to find fitness measures of a member of a mating pool population,
    #determined by the given benchmark test function
    inverse_fitness = lambda string: test_function(*general_decoder(string, variable_length, domain_min, domain_max, len(alnum_set)))


    #print perfromance of poulation
    #header
    print("\nTested population size: ", pop_size, " Number of generations: ", number_of_generations)
    pad = str((4 + LENGTH_OF_DECIMAL) * int(var_string_length / variable_length))
    print(("\n{:<16s}{:<" + pad + "s}\t {:}").format("Generation", "Strongest Candidate", "Fitness"))
    print("="*80)

    #print off generational performances, avoiding repeat performance levels between contiguous generations
    last_fit_individual = fittest_individual = [] #coordinate values
    last_max_fit = 0 #fitness value
    new_fit = True #is the found fitness value different than the last
    max_fitness_list = [] #list of all found fitness values to be used for a graph
    global_max_found = (0, [], 0) #to record the overall best found fitness measure form all generations
    first_gen_repeat = last_gen_repeat = 0 #to keep track of how many generations have had repeated max_fitness measures

    for i in range(number_of_generations):
```

```
        
        #create new generation of population
        population = perform_reproduction(population, inverse_fitness)
        population = perform_mating(population)
        population = perform_mutations(population, probability_of_mutation, alnum_set)
        
        #determine the max fitness measure of this generation
        max_fitness = max(1/(inverse_fitness(m)+1) for m in population)
```

```
            max_fitness_list.append(max_fitness)

            #determine the string variable values that have max_fitness of this generation
            for m in population:
                if 1/(inverse_fitness(m)+1) == max_fitness:
                    fittest_individual = general_decoder(m, variable_length, domain_min, domain_max, len(alnum_set))

    #case: if this generation is the last, or it is more fit than its predecessor
        if (i == number_of_generations - 1) or not (fittest_individual == last_fit_individual):
            #case: if is the new fittest member after repeated max peformance
            if (first_gen_repeat != last_gen_repeat) and (last_gen_repeat - first_gen_repeat > 1) :
                print(("\n\tFor generation {} to {}, the max fitness level was {:."+str(LENGTH_OF_DECIMAL)+"f}.\n").format(first_gen_repeat, last_gen_repeat, last_max_fit))

            #print generation's performance results
            print(("{:<16d}[{:<" + pad + "s}]\t {:>}").format(i, " ".join([("{:." + str(LENGTH_OF_DECIMAL) + "f},").format(x) for x in fittest_individual]), max_fitness))

            last_fit_individual = fittest_individual
            last_max_fit = max_fitness

            #record the overall best found fitness measure form all generations
            if max_fitness > global_max_found[2]:
                global_max_found = ("Gen: " + str(i), fittest_individual, max_fitness)

            first_gen_repeat = last_gen_repeat = i
            new_fit = True

        #case: first repeat of same fittest member between generations
        elif last_fit_individual == fittest_individual:
            new_fit = False
            last_gen_repeat = i

    print("="*80)
    print("Highest fitness acheived by:\n", global_max_found)
    print("="*80)
    print("")
    plt.plot(max_fitness_list)
    plt.show()
```

```
#simple_genetic_algorithm.py

from typing import List
import random
from functions import general_decoder
import matplotlib.pyplot as plt

LENGTH_OF_DECIMAL=4

def initialize(pop_size: int, alnum_set: List[str], var_string_length) -> List[str]:
    """
Generates pop_size of random strings with characters of their alphanumeric character
set.

    Args:
        pop_size (int): Number of strings to be generated
        alnum_set (List[str]): Valid characters to genrate random string from
        var_string_length (int): Number of characters expected to be in each string
    Returns:
        (List[str]): List of pop_size random strings made with characters of their
        alphanumeric
                        character set
    """

    return ["".join(random.choices(alnum_set, k=var_string_length)) for _ in
range(pop_size)]


def perform_reproduction(population, inverse_fitness_func) -> List:
    """
    Takes in a population of strings and probabalistically candidates for mating  based
    on the
    relative fitness of each string (i.e. the more fit members of the population will
    be picked more often
    and will therefore make up a bigger portion of the mating pool).

    Args:
        population (List[str]): List of strings that make up the mating pool
        inverse_fitness_func (<function>): Benchmark function with known global minima;
        returned values
                                            closer to zero mean the given arguments are
                                            closer to optimum
                                Args:
                                    (str): Alphanumeric string representing number
                                    system value(s)
                                Returns:
                                    (float): floating point value between 0 to 1;
                                    results closer to zero
                                            are closer to optimization
    Returns:
    (List[str]): List of strings that make up the new mating pool generation 43
    """

    #compile a list of floating point values, numbers closer to 0 have a higher fitness
    inverse_fitnesses = [inverse_fitness_func(s) for s in population]
    min_inv_fit = min(inverse_fitnesses)
    max_inv_fit = max(inverse_fitnesses)

    #compile a list of inverse fitness values that are normalized to [0,1] with regard
```

```
to
    #the range of the population's fitness measures, numbers closer to 0 have a higher fitness
    inverse_fitnesses = [(X-min_inv_fit)/(max_inv_fit-min_inv_fit+1) for X in inverse_fitnesses]                #adding 1 to avoid division by zero

    #invert list of normalized inverse fitness values so that numbers closer to 1 have a higher fitness
```

```
    fitnesses = [1 / ((s)+1) for s in inverse_fitnesses]

    #sum up all found fitness measures
    total_fitness = sum(fitnesses)

    #compile a list of each fitness measure's proportion to the sum of all found fitness measures
    #of the mating pool; to be used as a probability that the condidate will mate
    probabilities_of_reproduction = [fit / total_fitness for fit in fitnesses]

    #create list of randomly chosen strings that are weight biased
    return random.choices(population=population, weights=probabilities_of_reproduction, k=len(population))


def perform_mating(population):
    """
    Takes a list of strings, returns a list of strings after potential mating operations.

    Args:
        population (List[str]): List of strings that make up the mating pool
    Returns:
        (List[str]): List of strings that make up the now potentially modified mating pool
    """

    new_pop = []

    while (len(population) > 0):
        s1 = population.pop(random.randint(0, len(population) - 1))
        s2 = population.pop(random.randint(0, len(population) - 1))

        s1p, s2p = crossover_pair(s1, s2)
        new_pop.append(s1p)
        new_pop.append(s2p)

    return new_pop


def crossover_pair(s1, s2):
    """
    Takes two strings for crossover, randomly determines a crossover point
    between 1 and len(s1)-1 then performs crossover of character data at crossover point.
    Assumes len(s1) = len(s2)

    Args:
        s1 (str): string for mating crossover
        s2 (str): string for mating crossover
    Returns:
        s1p (str): string from mating crossover
    s2p (str): string from mating crossover
    """
    crossover_point = random.randint(1, len(s1) - 2)
    s1p = s1[:crossover_point] + s2[crossover_point:]
    s2p = s2[:crossover_point] + s1[crossover_point:]

    return (s1p, s2p)
```

```
def perform_mutations(population, probability_of_mutation, alnum_set):
    """
Takes a list of strings, returns a list of strings after potential mutation
operation.

    Args:
        population (List[str]): List of strings that make up the mating pool
        probability_of_mutation (float): decimal number between 0 and 1 that represents
```

```
                                           the liklihood
                                                     that a character will mutate into another character from
                                                     its alphanumeric character set
        alnum_set (List[str]): Valid alphanumeric characters to genrate number-system value-strings from
    Returns:
    (List[str]): List of strings that make up a potentially mutated mating pool
    """

    return [attempt_mutation(s, probability_of_mutation, alnum_set) for s in population]


def attempt_mutation(s, probability_of_mutation, alnum_set):
    """
    Takes a string and probabilisticly performs character mutation

    Args:
        s (str): string to mutate
        probability_of_mutation (float): decimal number between 0 and 1 that represents the liklihood
                                                     that a character will mutate into another character from
                                                     its alphanumeric character set
        alnum_set (List[str]): Valid alphanumeric characters to genrate number-system value-strings from
    Returns:
        None
    """

    #bit flipping
    #on average, method 2 seems to outperform method 1

    # #method 1
    # #choose one bit randomly if random chance falls into the probability  that the string should mutate
    # if random.random() < probability_of_mutation:
    #     bit_to_flip = random.randint(0, len(s) - 1)
    #     s = list(s)
    #     s[bit_to_flip] = random.choice(alnum_set)
    #     return "".join(s)
    # else:
    #     return s

    #method 2
    #bit by bit, perform mutation if random chance falls into the probability  that the bit should mutate
    for i in range(len(s)):
        if random.random() < probability_of_mutation:
            s = s[:i] + str(alnum_set[random.randint(0, len(alnum_set)-1)]) + s[i + 1:]
    return s

# program starts here:
# ------------------------------------------------------------------------------------------------------------------------------------------

def SGA(test_function, pop_size, alnum_set, var_string_length, variable_length,
```

```
domain_min, domain_max,
    number_of_generations, probability_of_mutation):
    """
    Simple Genetic Algorithm that finds global minima of test functions through
    generational mating,
    reproduction and bit mutations while printing out each generation's performance
    results.
```

```
    Args:
        test_function (<function>): Benchmark function that has known optimum values
        (global min)
        pop_size (int): the number of strings to create for population
        alnum_set (List[str]): Valid alphanumeric characters to genrate number system
        value-strings from
        var_string_length (int): character length of string that contains one or more
        number system value-string string variables
        variable_length (int): character length of one number system value-string
        variable
        domain_min (Union[float,int]): min value of operational domain
        domain_max (Union[float,int]): max value of operational domain
        number_of_generations (int): number of times to mate / mutate the poulation in
        the attempt to hone in
                                        on the optimal input values for the given
                                        test_function
        probability_of_mutation (float): decimal number between 0 and 1 that represents
        the liklihood
                                             that a character will mutate into another
                                             character from
                                             its alphanumeric character set
    Prints:
        table: generational performances, avoiding repeat max performance levels
        between contiguous generations
    """

    #initialize a random population of pop_size values to be the starting point for
    optimization attempt
    #returns string with (var_string_length * pop_size) number of characters
    population = initialize(pop_size, alnum_set, var_string_length)

    #small anonymous function used to find fitness measures of a member of a mating
    pool population,
    #determined by the given benchmark test function
    inverse_fitness = lambda string: test_function(*general_decoder(string,
    variable_length, domain_min, domain_max, len(alnum_set)))


    #print perfromance of poulation
    #header
    print("\nTested population size: ", pop_size, " Number of generations: ",
    number_of_generations)
    pad = str((4 + LENGTH_OF_DECIMAL) * int(var_string_length / variable_length))
    print(("\n{:<16s}{:<" + pad + "s}\t {:}").format("Generation", "Strongest
    Candidate", "Fitness"))
    print("="*80)

    #print off generational performances, avoiding repeat performance levels between
    contiguous generations
    last_fit_individual = fittest_individual = [] #coordinate values
    last_max_fit = 0 #fitness value
    new_fit = True #is the found fitness value different than the last
    max_fitness_list = [] #list of all found fitness values to be used for a graph
    global_max_found = (0, [], 0) #to record the overall best found fitness measure
    form all generations
    first_gen_repeat = last_gen_repeat = 0 #to keep track of how many generations have
    had repeated max_fitness measures

    for i in range(number_of_generations):
```

```
        #create new generation of population
        population = perform_reproduction(population, inverse_fitness)
        population = perform_mating(population)
        population = perform_mutations(population, probability_of_mutation, alnum_set)

        #determine the max fitness measure of this generation
        max_fitness = max(1/(inverse_fitness(m)+1) for m in population)
```

```
            max_fitness_list.append(max_fitness)
            #determine the string variable values that have max_fitness of
                this generation
            for m in population:
                if 1/(inverse_fitness(m)+1) == max_fitness:
                    fittest_individual = general_decoder(m,
                    variable_length, domain_min, domain_max,
                    len(alnum_set))

        #case: if this generation is the last, or it is more fit than its
        predecessor
            if (i == number_of_generations - 1) or not (fittest_individual
                    ==
            last_fit_individual):
                #case: if is the new fittest member after repeated max
                peformance
                if (first_gen_repeat != last_gen_repeat) and
                (last_gen_repeat -
                first_gen_repeat > 1) :
                    print(("\n\tFor generation {} to {}, the max fitness
                    level was
                    {:."+str(LENGTH_OF_DECIMAL)+"f}.\n").format(first
                    _gen_repeat, last_gen_repeat, last_max_fit))

                #print generation's performance results
                print(("{:<16d}[{:<" + pad + "s}]\t {:>}").format(i,
                " ".join([("{:." + str(LENGTH_OF_DECIMAL) +
                "f},").format(x) for x in fittest_individual]),
                max_fitness))

                last_fit_individual = fittest_individual
                last_max_fit = max_fitness
                #record the overall best found fitness measure form all
generations
                if max_fitness > global_max_found[2]:
                    global_max_found = ("Gen: " + str(i),
fittest_individual, max_fitness)
                first_gen_repeat = last_gen_repeat = i
                new_fit = True

            #case: first repeat of same fittest member between generations
            elif last_fit_individual == fittest_individual:
                new_fit = False
                last_gen_repeat = i

        print("="*80)
        print("Highest fitness acheived
by:\n", global_max_found)
        print("="*80)
        print("")
        plt.plot(max_fitness_list)
        plt.show()
```

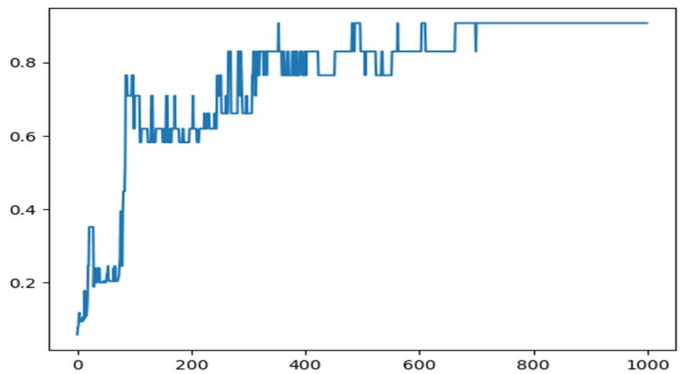
```
#dejong_test.py

from simple_genetic_algorithm import SGA

ALNUM = ["0", "1"]
VAR_STRING_LEN = 20
NUMBER_OF_VARIABLES = 4
VARIABLE_LEN = int(VAR_STRING_LEN / NUMBER_OF_VARIABLES)
PROBABILITY_OF_MUTATION = 0.05
POP_SIZE = 40
NUM_GENERATIONS = 1000
DOMAIN_MIN = -5.12
DOMAIN_MAX = 5.12

def dejong(*xs):
    """
    The function is defined on n-dimensional space.
    The function can be defined on any input domain but it is usually evaluated on
    x_i element of [-5.12, 5.12] for i = 1, ..., n.

    Takes in n dimensional coordinates and returns output of:
    f(x,y) = sum[b(x_i + 1 - (x_i)^2)^2 + (a - x_i)^2]]
    for i = 1, ..., n; and the parameters a and b are constants set to
    a = 1 and b = 100..

    The function has one global minimum f(x^*) = 0 at x^* = (0, ..., 0).

    Args:
        xs (List[num]): n dimensional coordinates for Euclidean (n + 1)-space
    Returns:
        (float): f(x,y) = value closer to 0 indicates a coordinate closer to know global minimum
    """

    return sum(xi ** 2 for xi in xs)

print("Running Simple Genetic Algorithm on De Jong Sphere benchmark function")
SGA(dejong, POP_SIZE, ALNUM, VAR_STRING_LEN, VARIABLE_LEN, DOMAIN_MIN, DOMAIN_MAX, NUM_GENERATIONS,
    PROBABILITY_OF_MUTATION)

print("\nDe Jong benchmark test complete\n")
```

```python
#himmelblau_test.py

from simple_genetic_algorithm import SGA

ALNUM = ["0", "1"]
VAR_STRING_LEN = 16
NUMBER_OF_VARIABLES = 2
VARIABLE_LEN = int(VAR_STRING_LEN / NUMBER_OF_VARIABLES)
PROBABILITY_OF_MUTATION = 0.005
POP_SIZE = 100
NUM_GENERATIONS = 1000
DOMAIN_MIN = -6
DOMAIN_MAX = 6

def himmelblau(*xs):
    """
    The function is defined on the 2-dimensional space.
    The function can be defined on any input domain but it is usually evaluated on
    x_i element of [-6, 6] for i = 1, 2.

    Takes in x, y cartesian values and returns function output of:
        f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2

    The function has four local minima at:
        f(x^*) = 0 at x^* = (3, 2)
        f(x^*) = 0 at x^* = (-2.805118, 3.283186)
        f(x^*) = 0 at x^* = (-3.779310, -3.283186)
        f(x^*) = 0 at x^* = (3.584458, -1.848126)

    Args:
        xs (List[num]): [x,y] cartesian coordinates
    Returns:
        (float): f(x,y) = value closer to 0 indicates a coordinate closer to know
        global minimum
    """

    x = xs[0]
    y = xs[1]
    return (x ** 2 + y - 11) ** 2 + (x + y ** 2 - 7) ** 2

print("Running Simple Genetic Algorithm on Himmelblau benchmark function")
SGA(himmelblau, POP_SIZE, ALNUM, VAR_STRING_LEN, VARIABLE_LEN, DOMAIN_MIN, DOMAIN_MAX,
NUM_GENERATIONS,
    PROBABILITY_OF_MUTATION)

print("\nHimmelblau benchmark test complete\n")
```

```python
#rosenbrock_test.py
from simple_genetic_algorithm import SGA

ALNUM = ["0", "1"]
VAR_STRING_LEN = 16
NUMBER_OF_VARIABLES = 2
VARIABLE_LEN = int(VAR_STRING_LEN / NUMBER_OF_VARIABLES)
PROBABILITY_OF_MUTATION = 0.01
POP_SIZE = 100
NUM_GENERATIONS = 1000
DOMAIN_MIN = -2
DOMAIN_MAX = 2


def rosenbrock(*xs):
    """
    The function is defined on n-dimensional space.
    The function can be defined on any input domain but it is usually evaluated on
    x_i element of [-5, 10] for i = 1, ..., n.

    Takes in n dimensional coordinates and returns output of:
        f(x,y) = sum[b(x_i + 1 - (x_i)^2)^2 + (a - x_i)^2]]
    for i = 1, ..., n; and the parameters a and b are constants set to a = 1 and b = 100..

    The function has one global minimum f(x^*) = 0 at x^* = (1, ..., 1).

    Args:
        xs (List[num]): n dimensional coordinates for Euclidean (n + 1)-space
    Returns:
        (float): f(x,y) = value closer to 0 indicates a coordinate closer to know global minimum
    """

    x = xs[0]
    y = xs[1]

    return (1-x)**2+100*(y-x**2)**2


print("Running Simple Genetic Algorithm on Rosenbrock benchmark function")

SGA(rosenbrock, POP_SIZE, ALNUM, VAR_STRING_LEN, VARIABLE_LEN, DOMAIN_MIN, DOMAIN_MAX, NUM_GENERATIONS,
    PROBABILITY_OF_MUTATION)

print("\nRosenbrock benchmark test complete\n")
```

De Jong

Method 1

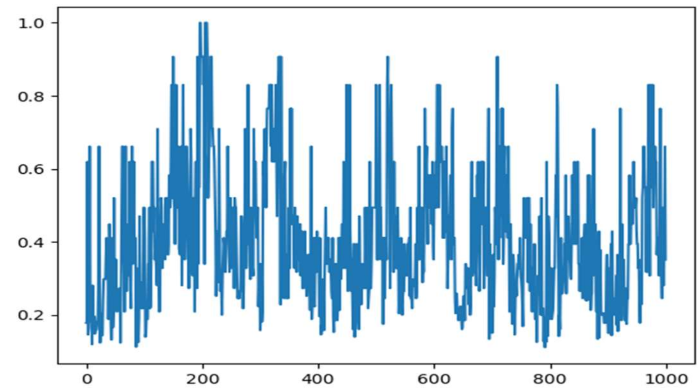



```
Running Simple Genetic Algorithm on De Jong Sphere benchmark function

Tested population size:  40  Number of generations:  1000

Generation      Strongest Candidate                             Fitness
==========================================================================
0               [-1.9200, 1.9200, -1.9200, 2.2400,]             0.05855898060526562
1               [-1.6000, 1.6000, 2.5600, -0.3200,]             0.07827175954915469
2               [-1.9200, 1.9200, -1.9200, 0.6400,]             0.08020017964840241
3               [-0.3200, -1.9200, -1.9200, 0.6400,]            0.11255177381595535
4               [-0.3200, -1.9200, -1.9200, 0.0000,]            0.11799131583915423
5               [-1.6000, -1.6000, -1.9200, 0.9600,]            0.09321401938851603
6               [-1.2800, 1.9200, -1.9200, 0.0000,]             0.09988812529966437
7               [-1.2800, 2.2400, -1.6000, -0.3200,]            0.0969142502713599
9               [-1.2800, 0.6400, 2.2400, -1.6000,]             0.09411233248004816
10              [-0.6400, 0.6400, 2.2400, -1.6000,]             0.10641920653839604
12              [-2.5600, 0.6400, 0.6400, 1.2800,]              0.09988812529966438
13              [-1.2800, 0.9600, -0.6400, 1.2800,]             0.17831669044222537
14              [-1.2800, 0.9600, 0.9600, -2.2400,]             0.10527202290719216
15              [-0.6400, 1.6000, -1.6000, 1.6000,]             0.11001584228128854
17              [-0.6400, 0.9600, 0.9600, -2.2400,]             0.12091313600309538
18              [-1.2800, 0.9600, 0.6400, 1.6000,]              0.15314873805439846
19              [-1.2800, 0.9600, 0.6400, 0.3200,]              0.24557956777996065
21              [-0.6400, 0.9600, 0.6400, 0.3200,]              0.35171637591446264

        For generation 21 to 28, the max fitness level was 0.3517.

29              [-0.6400, 0.9600, 0.6400, 1.6000,]              0.18865076969514044
```

…

```
552             [-0.3200, 0.0000, 0.3200, 0.0000,]              0.8300132802124832

        For generation 552 to 561, the max fitness level was 0.8300.

562             [-0.3200, 0.0000, 0.0000, 0.0000,]              0.9071117561683597
563             [-0.3200, 0.0000, 0.3200, 0.0000,]              0.8300132802124832

        For generation 563 to 603, the max fitness level was 0.8300.

604             [-0.3200, 0.0000, 0.0000, 0.0000,]              0.9071117561683597

        For generation 604 to 610, the max fitness level was 0.9071.

611             [-0.3200, 0.0000, 0.3200, 0.0000,]              0.8300132802124832

        For generation 611 to 662, the max fitness level was 0.8300.

663             [-0.3200, 0.0000, 0.0000, 0.0000,]              0.9071117561683597

        For generation 663 to 698, the max fitness level was 0.9071.

699             [-0.3200, 0.0000, 0.3200, 0.0000,]              0.8300132802124832
700             [-0.3200, 0.0000, 0.0000, 0.0000,]              0.9071117561683597

        For generation 700 to 998, the max fitness level was 0.9071.

999             [-0.3200, 0.0000, 0.0000, 0.0000,]              0.9071117561683597
==========================================================================
Highest fitness acheived by:
 ('Gen: 353', [-0.3200000000000003, 0.0, 0.0, 0.0], 0.9071117561683597)
==========================================================================


De Jong benchmark test complete
```

Method 2



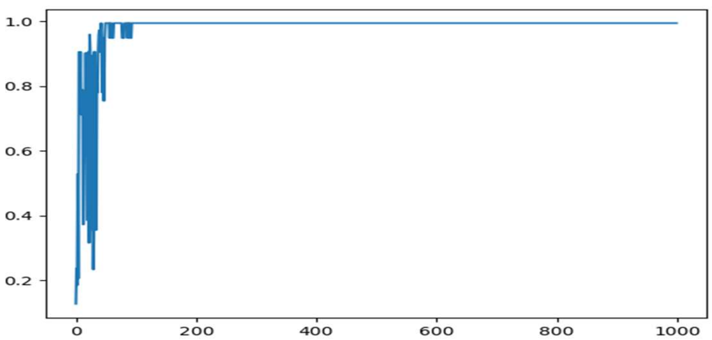

```
Running Simple Genetic Algorithm on De Jong Sphere benchmark function

Tested population size:  40  Number of generations:  1000

Generation      Strongest Candidate                             Fitness
==========================================================================
0               [0.6400, -1.2800, 0.0000, 1.6000,]              0.17831669044222542
1               [1.2800, -0.6400, 0.0000, 1.6000,]              0.17831669044222542
2               [0.6400, -0.3200, 0.0000, 0.3200,]              0.6194251734390486
3               [0.6400, -1.2800, 1.9200, 0.3200,]              0.146267259536253
4               [-0.6400, -0.3200, -0.9600, 1.9200,]            0.16339869281045755
5               [1.2800, 0.6400, 0.0000, 0.9600, ]              0.25191455058444173
6               [0.0000, -0.6400, 0.0000, 0.3200,]              0.6613756613756615
7               [0.0000, -0.6400, 1.9200, 0.3200,]              0.19236688211757466
8               [0.0000, -1.9200, -0.6400, -0.6400,]            0.18163324614937523
9               [0.0000, -1.9200, -0.6400, -0.3200,]            0.19236688211757466
10              [0.0000, -2.5600, 0.6400, -0.6400,]             0.11943435887636158
11              [0.6400, 0.6400, -1.2800, 0.3200,]              0.2808988764044944
12              [0.0000, -1.9200, 0.6400, -0.9600,]             0.1661792076575379
13              [0.9600, -1.9200, 0.0000, -0.9600,]             0.15314873805439844
14              [0.0000, -1.9200, 0.0000, -0.6400,]             0.19623233908948198
15              [1.2800, -1.9200, 0.0000, -0.6400,]             0.14849132810643859
16              [0.6400, -1.9200, 0.0000, -0.6400,]             0.18163324614937523
17              [1.2800, -1.2800, 1.2800, -0.6400,]             0.15810776625347833
18              [1.2800, -1.2800, 1.2800, -0.3200,]             0.16617920765753783
19              [0.6400, -1.9200, -0.6400, -0.3200,]            0.17831669044222542
20              [-0.6400, 0.6400, -1.2800, 1.2800,]             0.19623233908948196
21              [-0.6400, -1.2800, 0.0000, -0.6400,]            0.28921795465062466
```

…

```
966             [0.0000, -0.6400, 0.3200, -1.2800,]             0.3174200101574403
967             [0.0000, -0.3200, 0.6400, -0.6400,]             0.5203996669442134
968             [1.2800, -0.6400, 0.0000, -0.3200,]             0.31742001015744026
969             [0.0000, -0.3200, 0.0000, -0.6400,]             0.6613756613756615
970             [0.0000, -0.3200, 0.0000, -0.3200,]             0.8300132802124832
971             [0.6400, -0.3200, 0.3200, -0.3200,]             0.5824790307548928
972             [1.2800, -0.6400, 0.3200, -0.3200,]             0.307427447122479
973             [0.0000, 0.0000, 0.3200, -0.3200,]              0.8300132802124832
974             [0.0000, -0.3200, 0.0000, -0.3200,]             0.8300132802124832
975             [0.0000, -0.6400, 0.0000, -0.6400,]             0.5496921723834656
976             [0.0000, -0.3200, 0.6400, -0.3200,]             0.6194251734390486
977             [0.0000, -0.3200, 0.0000, -0.3200,]             0.8300132802124832
978             [0.6400, -0.3200, 0.6400, -0.3200,]             0.4940711462450594
979             [0.0000, -0.3200, 0.3200, -0.6400,]             0.6194251734390486
981             [0.0000, -0.3200, 0.0000, -0.6400,]             0.6613756613756615
982             [0.0000, -0.3200, 0.3200, -0.6400,]             0.6194251734390486
984             [0.0000, -0.9600, 0.6400, -0.6400,]             0.3648569760653825
985             [0.0000, -0.3200, 0.9600, -0.6400,]             0.410913872452334
986             [0.0000, -1.2800, 0.0000, -0.3200,]             0.36485697606538225
988             [0.0000, -0.9600, 0.9600, -0.6400,]             0.3074274471224791
989             [0.6400, 0.6400, 0.0000, -0.6400,]              0.4486719310839916
990             [0.0000, -0.3200, 0.3200, -0.3200,]             0.7649938800489593
991             [1.2800, -0.3200, 0.0000, -0.6400,]             0.31742001015744026
993             [1.2800, -0.3200, 0.6400, -0.9600,]             0.24557956777996065
994             [0.3200, -0.3200, 0.6400, -0.6400,]             0.4940711462450594
995             [0.0000, -0.3200, 0.6400, -0.9600,]             0.410913872452334
996             [1.2800, -0.6400, 0.3200, -0.6400,]             0.28089887640449435
997             [0.0000, -0.6400, 0.0000, -0.9600,]             0.42896362388469467
998             [0.0000, -0.6400, 0.0000, -0.3200,]             0.6613756613756615
999             [0.6400, -0.3200, 0.6400, -0.9600,]             0.35171637591446264
==========================================================================
Highest fitness acheived by:
 ('Gen: 196', [0.0, 0.0, 0.0, 0.0], 1.0)
==========================================================================


De Jong benchmark test complete
```

Himmelblau

Method 1

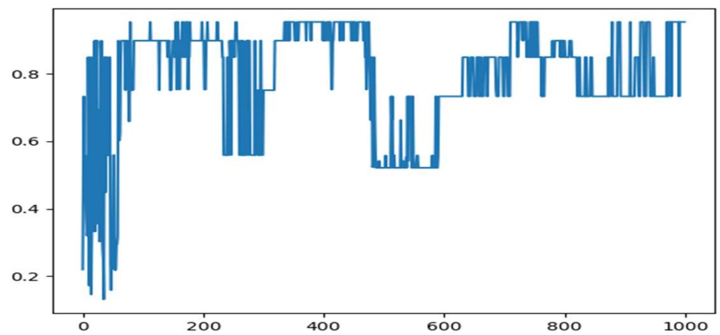



```
Running Simple Genetic Algorithm on Himmelblau benchmark function

Tested population size:  100  Number of generations:  1000

Generation        Strongest Candidate          Fitness
=======================================================================
0                 [3.4219, -0.9844,]           0.12685092257472316
1                 [3.4219, -2.1562,]           0.23576390181437498
2                 [-2.4375, 3.0000,]           0.18418870798658832
3                 [3.0469, 1.7344, ]           0.5281303795684011
4                 [-2.4375, 3.0938,]           0.2049805991176992
5                 [-2.7656, 3.0938,]           0.9057085681453921
7                 [-2.7656, 3.0469,]           0.7546138259377453
8                 [-2.7656, 3.0938,]           0.9057085681453921
9                 [-2.9062, 3.0938,]           0.7123313727807917
10                [3.5625, -1.9688,]           0.7879095273411465
12                [3.5625, -2.1562,]           0.3724650854831554
14                [3.5625, -2.0625,]           0.5542531418616059
15                [3.5625, -2.0156,]           0.6684277361309007
16                [3.6094, -1.7812,]           0.9024394166606019
17                [3.5625, -2.0156,]           0.6684277361309007
19                [3.6094, -1.7812,]           0.9024394166606019
20                [3.6094, -2.1562,]           0.38445216363670154
21                [3.6094, -1.9219,]           0.9066419898073936
22                [3.6094, -2.2031,]           0.31530059654322834
23                [3.6094, -1.8281,]           0.9595585892857286
24                [3.6094, -1.7812,]           0.9024394166606019
25                [3.6094, -2.0156,]           0.6887587173430882
26                [3.5625, -1.4531,]           0.35525716342591057
```

…

```
57                [-2.8125, 3.1406,]           0.9948443921995289

        For generation 57 to 59, the max fitness level was 0.9948.

60                [-2.7656, 3.1406,]           0.9487880473133589

        For generation 60 to 62, the max fitness level was 0.9488.

63                [-2.8125, 3.1406,]           0.9948443921995289

        For generation 63 to 76, the max fitness level was 0.9948.

77                [-2.7656, 3.1406,]           0.9487880473133589

        For generation 77 to 80, the max fitness level was 0.9488.

81                [-2.8125, 3.1406,]           0.9948443921995289

        For generation 81 to 85, the max fitness level was 0.9948.

86                [-2.7656, 3.1406,]           0.9487880473133589
87                [-2.8125, 3.1406,]           0.9948443921995289
89                [-2.7656, 3.1406,]           0.9487880473133589
90                [-2.8125, 3.1406,]           0.9948443921995289
91                [-2.7656, 3.1406,]           0.9487880473133589
93                [-2.8125, 3.1406,]           0.9948443921995289

        For generation 93 to 998, the max fitness level was 0.9948.

999               [-2.8125, 3.1406,]           0.9948443921995289
=======================================================================
Highest fitness acheived by:
 ('Gen: 42', [-2.8125, 3.140625], 0.9948443921995289)
=======================================================================


Himmelblau benchmark test complete
```

Method 2



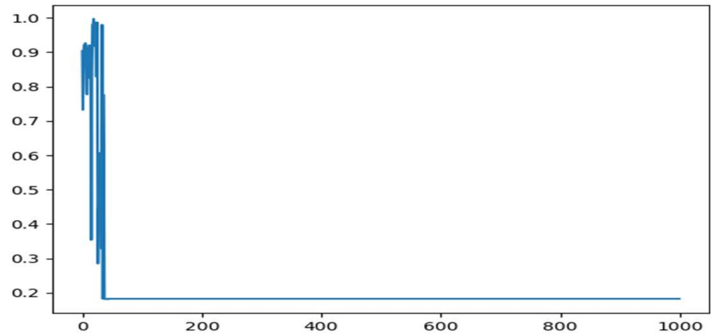

```
Running Simple Genetic Algorithm on Himmelblau benchmark function

Tested population size:  100  Number of generations:  1000

Generation        Strongest Candidate          Fitness
=======================================================================
0                 [3.1875, 2.2500, ]           0.21972996442665219
1                 [-3.6562, -3.1875,]          0.521534039504397
2                 [-3.7500, -3.1875,]          0.7333848099282685
3                 [-3.7500, -3.0938,]          0.41835122118981877
4                 [-3.7500, -3.1406,]          0.5580541659818725
5                 [-2.7656, 2.9531,]           0.44446231772630235
6                 [3.0000, 2.2969, ]           0.3682727405489496
7                 [3.1406, 2.2031, ]           0.3198419604143811
8                 [3.2344, 1.9219, ]           0.3426966317380834
9                 [-3.7500, -3.3281,]          0.8495425659298472
10                [3.0000, 2.2031, ]           0.5649189289312718
11                [-3.0469, 2.8594,]           0.17223202916858715
12                [-2.8125, 2.9531,]           0.452451961679104
13                [-3.7500, -3.3281,]          0.8495425659298472
14                [2.5312, 2.2031, ]           0.14580124120040613
15                [3.7031, -1.5469,]           0.3147154801968908
16                [3.6562, -2.1094,]           0.43676270558425573
17                [-3.7500, -3.3281,]          0.8495425659298472
18                [3.4688, -2.1094,]           0.3329600109200428
19                [-3.7500, -3.3281,]          0.8495425659298472
20                [-3.7500, -3.2344,]          0.8985072610951793
21                [3.4688, -2.1094,]           0.3329600109200428
```

…

```
        For generation 936 to 941, the max fitness level was 0.9541.

942               [-3.7500, -3.1875,]          0.7333848099282685

        For generation 942 to 946, the max fitness level was 0.7334.

947               [-3.7500, -3.3281,]          0.8495425659298472
949               [-3.7500, -3.1875,]          0.7333848099282685

        For generation 949 to 969, the max fitness level was 0.7334.

970               [-3.7500, -3.2812,]          0.9540826957320063
971               [-3.7500, -3.1875,]          0.7333848099282685
972               [-3.7500, -3.2812,]          0.9540826957320063

        For generation 972 to 976, the max fitness level was 0.9541.

977               [-3.7500, -3.1875,]          0.7333848099282685
978               [-3.7500, -3.2812,]          0.9540826957320063

        For generation 978 to 989, the max fitness level was 0.9541.

990               [-3.7500, -3.1875,]          0.7333848099282685
991               [-3.7500, -3.2812,]          0.9540826957320063

        For generation 991 to 998, the max fitness level was 0.9541.

999               [-3.7500, -3.2812,]          0.9540826957320063
=======================================================================
Highest fitness acheived by:
 ('Gen: 79', [-3.75, -3.28125], 0.9540826957320063)
=======================================================================


Himmelblau benchmark test complete
```

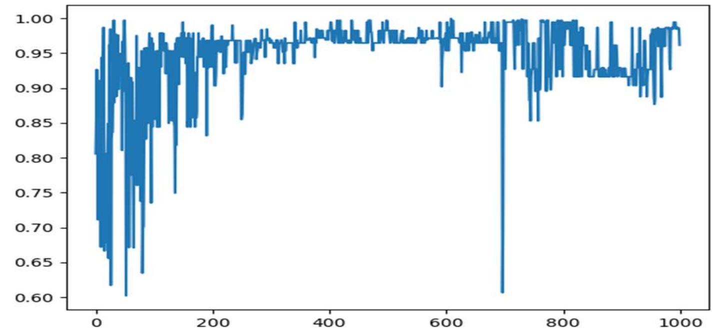
Rosenbrock

Method 1




```
Running Simple Genetic Algorithm on Rosenbrock benchmark function

Tested population size:  100  Number of generations:  1000

Generation       Strongest Candidate        Fitness
===========================================================================
0                [1.2031, 1.4219, ]         0.9033636766002906
1                [0.9531, 0.9688, ]         0.7321505844648348
2                [0.6719, 0.4531, ]         0.9025612245710402
3                [1.2500, 1.5781, ]         0.9200359389038635
4                [1.2500, 1.5469, ]         0.9200359389038635
5                [0.7188, 0.5156, ]         0.9266149414114279
6                [0.7500, 0.5781, ]         0.9200359389038635
7                [0.7500, 0.5156, ]         0.7798933739527799
8                [0.9219, 0.7969, ]         0.7771361403442459
9                [0.9844, 0.9375, ]         0.9095603828204
10               [0.7969, 0.5938, ]         0.8254253515527755
11               [0.7500, 0.5781, ]         0.9200359389038635
12               [0.7500, 0.5469, ]         0.9200359389038635
14               [0.9375, 0.9219, ]         0.8413701021927797
15               [0.7344, 0.6719, ]         0.35360758557227123
16               [0.9375, 0.9219, ]         0.8413701021927797
17               [0.9531, 0.9219, ]         0.9801733757561382
19               [0.9531, 0.9062, ]         0.9973271002925899
20               [0.8906, 0.7656, ]         0.9190567751421268
21               [0.9531, 0.9219, ]         0.9801733757561382
22               [0.8906, 0.7969, ]         0.9868706773126782
23               [0.5469, 0.2969, ]         0.8293214471223307
24               [0.8750, 0.7969, ]         0.8982456140350877
25               [0.8906, 0.7969, ]         0.9868706773126782
26               [0.9531, 0.7500, ]         0.2846771771491518
27               [0.9531, 0.7969, ]         0.44503103266984945
29               [0.9531, 0.7812, ]         0.38166311708082545
30               [0.9531, 0.8281, ]         0.6070304195138754
32               [0.9531, 0.7656, ]         0.328729236951222
33               [0.9531, 0.9219, ]         0.9801733757561382
34               [-1.1094, 1.2500,]         0.18226017435828668
35               [0.9219, 0.7969, ]         0.7771361403442459
36               [0.9219, 0.7656, ]         0.5829041895321386
37               [-1.1094, 1.2500,]         0.18226017435828668
38               [-1.1250, 1.2656,]         0.1813031161473088

        For generation 38 to 42, the max fitness level was 0.1813.

43               [-1.1094, 1.2500,]         0.18226017435828668
44               [-1.1250, 1.2656,]         0.1813031161473088
45               [-1.1094, 1.2500,]         0.18226017435828668

        For generation 45 to 998, the max fitness level was 0.1823.

999              [-1.1094, 1.2500,]         0.18226017435828668
===========================================================================
Highest fitness acheived by:
 ('Gen: 19', [0.953125, 0.90625], 0.9973271002925899)
===========================================================================


Rosenbrock benchmark test complete
```

Method 2




```
Running Simple Genetic Algorithm on Rosenbrock benchmark function

Tested population size:  100  Number of generations:  1000

Generation       Strongest Candidate        Fitness
===========================================================================
0                [0.5781, 0.3594, ]         0.8056634380357375
1                [0.7188, 0.5469, ]         0.8541535654355582
2                [0.7188, 0.5156, ]         0.9266149414114279
3                [0.6094, 0.3906, ]         0.8404864689282009
4                [1.1250, 1.2031, ]         0.7111111111111111
5                [0.6250, 0.3438, ]         0.7351040918880115
6                [1.2969, 1.6719, ]         0.9106188394120442

        For generation 6 to 8, the max fitness level was 0.9106.

9                [0.3750, 0.1719, ]         0.6719160104986877
10               [1.0938, 1.2188, ]         0.9440745049032495
11               [0.8750, 0.7969, ]         0.8982456140350877
12               [0.8906, 0.7656, ]         0.9190567751421268
13               [0.8906, 0.7969, ]         0.9868706773126782
15               [0.4219, 0.2188, ]         0.6664623251239198
16               [0.5312, 0.2969, ]         0.8056821639430923

        For generation 16 to 18, the max fitness level was 0.8057.

19               [0.3125, 0.0938, ]         0.6783422349190577
20               [0.5000, 0.2344, ]         0.7846743295019157
```

...

```
969              [0.8750, 0.7656, ]         0.9846153846153847
971              [0.9375, 0.8594, ]         0.9596438821531073
972              [0.8750, 0.7656, ]         0.9846153846153847
974              [0.9375, 0.8594, ]         0.9596438821531073
975              [0.9062, 0.8281, ]         0.9867167533019916

        For generation 975 to 979, the max fitness level was 0.9867.

980              [0.8750, 0.7656, ]         0.9846153846153847
981              [0.9062, 0.8281, ]         0.9867167533019916
982              [0.9062, 0.7969, ]         0.9359845468859904
983              [0.7188, 0.5156, ]         0.9266149414114279
984              [0.8750, 0.7656, ]         0.9846153846153847
986              [0.8906, 0.7969, ]         0.9868706773126782
987              [0.8750, 0.7656, ]         0.9846153846153847
989              [0.9375, 0.8750, ]         0.9945972196928307

        For generation 989 to 992, the max fitness level was 0.9946.

993              [0.8750, 0.7656, ]         0.9846153846153847

        For generation 993 to 995, the max fitness level was 0.9846.

996              [0.8906, 0.7969, ]         0.9868706773126782
998              [0.8750, 0.7656, ]         0.9846153846153847
999              [0.8750, 0.7500, ]         0.9615023474178404
===========================================================================
Highest fitness acheived by:
 ('Gen: 608', [0.984375, 0.96875], 0.9997499614454374)
===========================================================================


Rosenbrock benchmark test complete
```