IQRA University

CSC 471 Artificial Intelligence

Lab# 07
Evolutionary Algorithms

Objective:

This lab introduces the students to the concept of meta-heuristic search. The lab also provides a stepwise implementation of Evolutionary algorithms to solve basic optimization problems

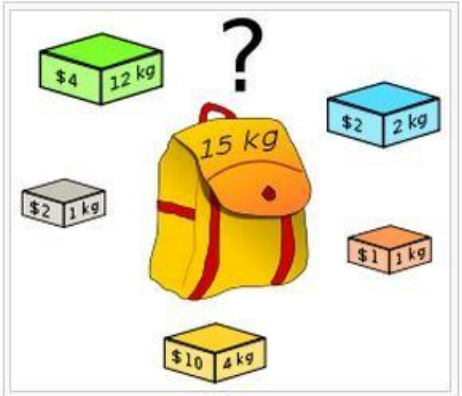
Name of Student: FARAZ ALAM

Roll No: 13948 **Sec.** Saturday(12:00-14:00)

Date of Experiment: 17/12/23

Lab 07: Optimization Problems

Optimization problems are a fundamental concept in artificial intelligence, aiming to find the best solution from a set of possible solutions, often within a complex and dynamic environment. These problems arise in various AI applications, from route planning and resource allocation to game playing and machine learning.



At their core, optimization problems involve exploring the solution space to identify the most favorable outcome based on a set of criteria or objectives. The goal is to efficiently navigate through the possibilities, ultimately converging on the solution that optimizes a given metric. This metric, often referred to as the "objective function," quantifies the quality of a solution.

Solving optimization search problems often involves striking a balance between exploration and exploitation, as the algorithm must navigate the trade-off between discovering new, potentially better solutions and refining existing ones. As AI continues to advance, optimization search techniques evolve, contributing to the development of more sophisticated and efficient algorithms that can tackle increasingly complex problems across diverse domains.

Certainly! Here's a text that you can use for an evolutionary algorithm lab, focusing on crossover, mutation, and survival selection for a phenotypic representation:

Introduction

Evolutionary Algorithms (EAs) are optimization and search techniques inspired by the process of natural selection. These algorithms are particularly effective in solving complex problems where the solution space is vast and poorly understood. In this lab, we will explore the implementation of Evolutionary Algorithms for solving a variety of problems. The lab will also delve into three fundamental components of EAs: Crossover, Mutation, and Survival Selection, with a focus on phenotypic representations.

Phenotypic Representation

In evolutionary algorithms, individuals are typically represented as phenotypes, where the phenotype is a solution to the optimization problem at hand. The phenotypic representation is crucial, as it determines how genetic operators like crossover and mutation manipulate potential solutions.

Crossover

Crossover, also known as recombination, is a genetic operator that combines genetic material from two parent individuals to create one or more offspring. The idea is to mimic the process of genetic recombination that occurs during sexual reproduction in nature.

For phenotypic representations, crossover involves exchanging parts of the phenotypic structures between parents. This process generates offspring with a combination of characteristics from both parents.

Mutation

Mutation is another genetic operator that introduces small, random changes in an individual's genetic material. This mimics the process of genetic mutations in nature, providing a way to explore new regions of the solution space.

In the context of phenotypic representations, mutation involves modifying certain components of the phenotype. This introduces variability in the population, preventing premature convergence to suboptimal solutions.

Survival Selection

After the application of genetic operators, a selection mechanism determines which individuals survive to the next generation. Survival selection is crucial for shaping the evolutionary process, favoring individuals with higher fitness values.

In phenotypic representations, survival selection is based on the performance of individuals in the given optimization problem. Individuals with better phenotypic traits (higher fitness) are more likely to survive and pass their traits to the next generation.

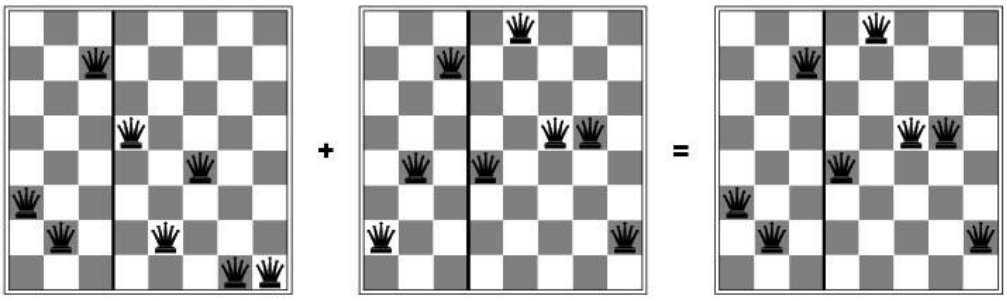
Evolutionary Algorithm Execution

The evolutionary algorithm drives biological evolution. The algorithm repeatedly modifies a population of individual solutions. At each step, the evolution selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety

of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear.

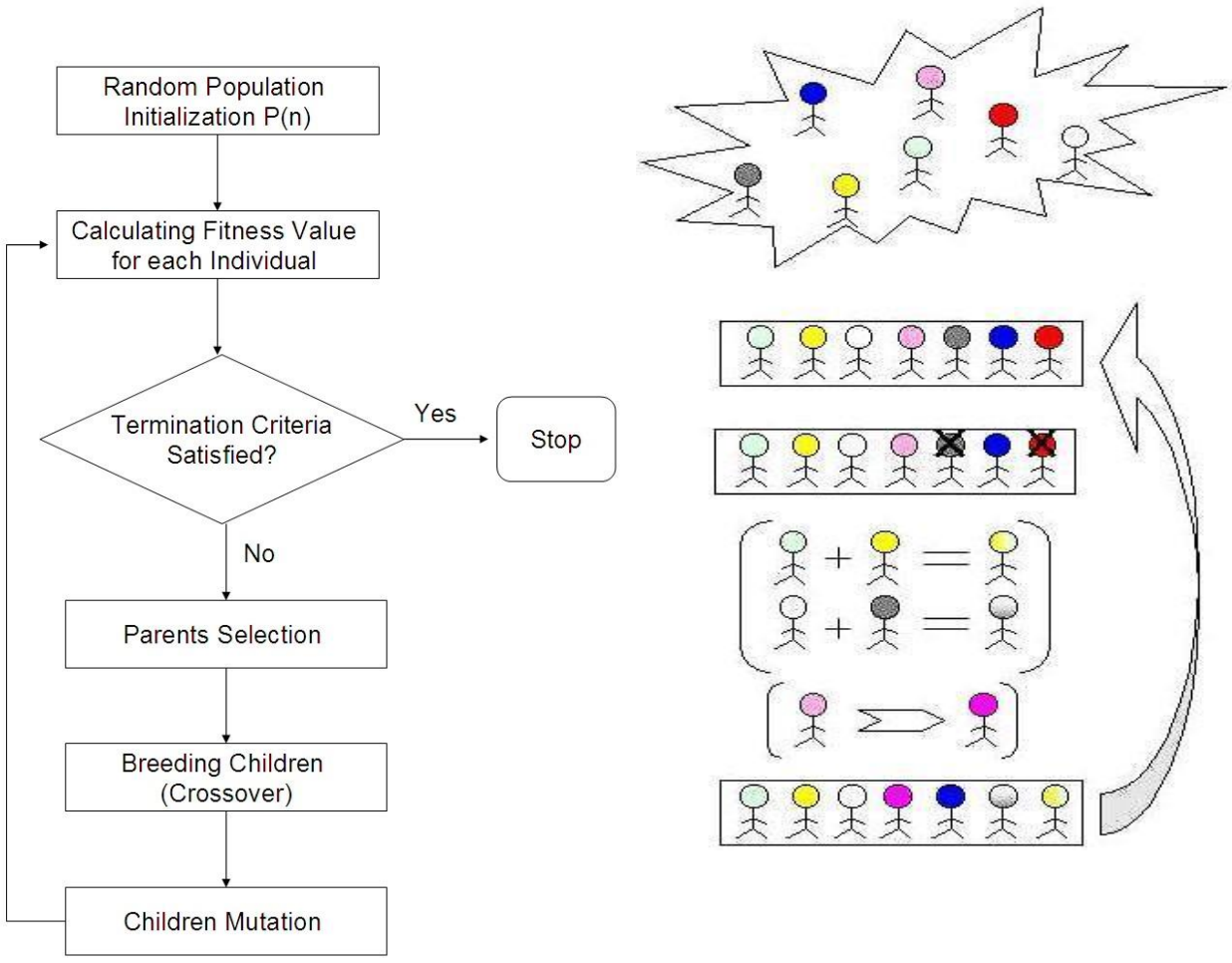
There are some key points to understand genetic algorithm:

- A successor state is generated by combining two parent states.
- Start with k randomly generated states (population).
- A state/individual is represented as a string over a finite alphabet (often a string of 0s and 1s).
- Evaluation function (fitness function). Higher values for better states.
- Produce the next generation of states by selection, crossover, and mutation.



Flowchart of EA

- all individuals in **population** evaluated by **fitness function**
- individuals allowed to **produce child states (selection), crossover, mutate**

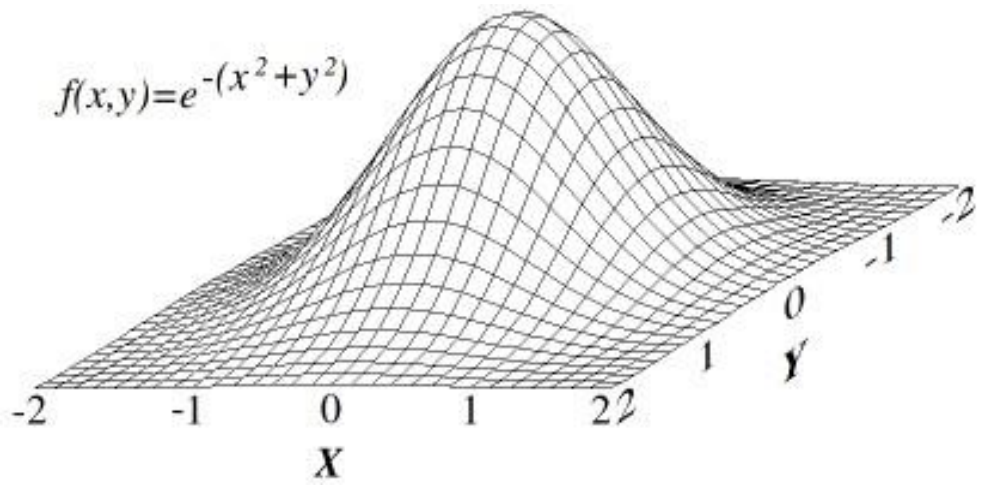


Algorithm

```
1)    Randomly initialize populations p
2)    Determine fitness of population  3) Until
convergence repeat:
a)  Select parents from population
b)  Crossover and generate new population
c)  Perform mutation on new population
d)  Calculate fitness for new population
```

Student Exercise

Task 1

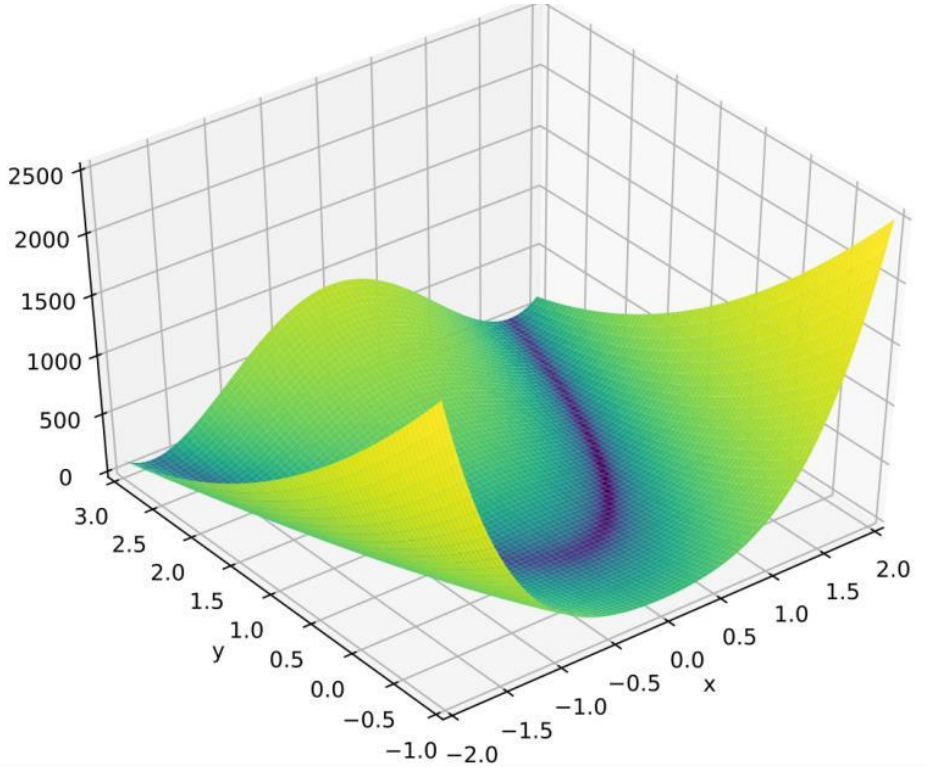


In [1]:

```python
import random
import copy from math import e

class chromosome:

    def __init__(self, x1, x2, y1, y2):
        self.x = random.uniform(x1,x2)
        self.y = random.uniform(y1,y2)
        self.fitness = self.evaluate()



    def evaluate(self):
        return (e**-((self.x**2) + (self.y**2)))


class selection():
    def binary_tournament(pop, T_size):
        tournament_teams = []
        for i in range(T_size):
            tournament_teams.append(random.choice(pop))
        return max(tournament_teams, key=lambda item: item.fitness)

class operators():
    def simple_crossover(parent1, parent2):
        child1 = copy.deepcopy(parent1)
        child2 = copy.deepcopy(parent2)
        child1.y = parent2.y
        child2.y = parent1.y

        child1.fitness = child1.evaluate()
        child2.fitness = child2.evaluate()
        return child1, child2

    def creep_mutation(offspring):
        print('before mutation', offspring.x, offspring.y, offspring.fitness)
        if random.random() < 0.5:
            if random.random() < 0.5:
                offspring.x = min(2, offspring.x+0.025)
            else:
                offspring.x = max(-2, offspring.x-0.025)
```

```
                offspring.x = max(-2, offspring.x-0.025)

        else:
            if random.random() < 0.5:
                offspring.y = min(2, offspring.y + 0.025)
            else:
                offspring.y = max(-2, offspring.y - 0.025)
        offspring.fitness = offspring.evaluate()
        print('after mutation', offspring.x, offspring.y, offspring.fitness)

pop = [_ for _ in range(10)]
for i in range(10):
    pop[i] = chromosome(-2,2,-2,2)


for iter in range(1000):
    new_population = []
    for i in range(5):
        for j in range(2):
            parent1 = selection.binary_tournament(pop, 2)
            parent2 = selection.binary_tournament(pop,2)

        offspring1, offspring2 = operators.simple_crossover(parent1, parent2)
        if random.random() < 0.2:
            if random.random() < 0.5:
                operators.creep_mutation(offspring1)
            else:
                operators.creep_mutation(offspring2)
        new_population.append(offspring1)
        new_population.append(offspring2)
    combined_pool = pop+ new_population
    print(len(combined_pool))

    # combined_pool.sort(key=lambda item: item.fitness, reverse = True)
    # for solution in combined_pool:
    #     print(solution.fitness)
    print('population before')
    for value in pop:
        print(value.x, value.y, value.fitness)
    for i in range(len(pop)):
        pop[i] = selection.binary_tournament(combined_pool, 10)

    print('population after')
    for value in pop:
        print(value.x, value.y, value.fitness)
pop.sort(key=lambda item: item.fitness)
best_solution = pop[-1]
print(best_solution.x, best_solution.y, best_solution.fitness)
```

```
before mutation 0.06878594812183847 0.23546610702498372 0.9415989913770273
after mutation 0.04378594812183847 0.23546610702498372 0.9442526633356103
20
population before
-0.8738333223521906 0.23546610702498372 0.4408595377335694
0.47290102731127037 1.5772592904987395 0.06644489967490476
0.37442136647846613 0.08406759633843208 0.8630706545178485
1.1501131976333716 -0.2714671316429871 0.24747275819434864
1.3550033082200739 -1.328278736996431 0.027313932241745102
0.06878594812183847 1.373847796830027 0.1507414327985098
0.8097992862857768 1.4425906950008245 0.06477453071732162
-1.6809467937588907 -1.11175572386328 0.017221737017195878
0.7645865805851191 1.9168506075892555 0.014137719790546708
-0.3634191823782027 0.9207535639821578 0.3753634104654126
population after
-0.3634191823782027 0.08406759633843208 0.8701054542699167
-0.8738333223521906 0.23546610702498372 0.4408595377335694
0.04378594812183847 0.23546610702498372 0.9442526633356103
0.04378594812183847 0.23546610702498372 0.9442526633356103
```



$$f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

Implement an Evolutionary Algorithm for finding the optimal solution to the given two function. The EA should provide values for the x and y parameters where the value of the function maximizes. The figures also provide the range of values the solution exist within. Restrict your search within the domain of the variables for a quicker response. Your implementation of the EA should include.

1. **At least 2 selection mechanisms (pre and post selection combined)**
2. **At least One crossover scheme**
3. **At least two mutation schemes**

In [2]:

```python
import random
import copy from math import e

class Chromosome:

    def __init__(self, x1, x2, y1, y2):
        self.x = random.uniform(x1, x2)
        self.y = random.uniform(y1, y2)
        self.fitness = self.evaluate()

    def evaluate(self):
        return (1 - self.x)**2 + 100 * (self.y - self.x**2)**2

class Selection:

    @staticmethod
    def binary_tournament(pop, T_size):
        tournament_teams = []
        for i in range(T_size):
            tournament_teams.append(random.choice(pop))
        return max(tournament_teams, key=lambda item: item.fitness)

    @staticmethod
    def roulette_wheel(population):
        fitness_sum = sum([ind.fitness for ind in population])
        selection_point = random.uniform(0, fitness_sum)
        current_sum = 0
        for individual in population:
            current_sum += individual.fitness
            if current_sum > selection_point:
                return individual

class Operators:

    @staticmethod
    def simple_crossover(parent1, parent2):
        child1 = copy.deepcopy(parent1)
        child2 = copy.deepcopy(parent2)
        child1.y = parent2.y
        child2.y = parent1.y
        child1.fitness = child1.evaluate()
        child2.fitness = child2.evaluate()
        return child1, child2

```

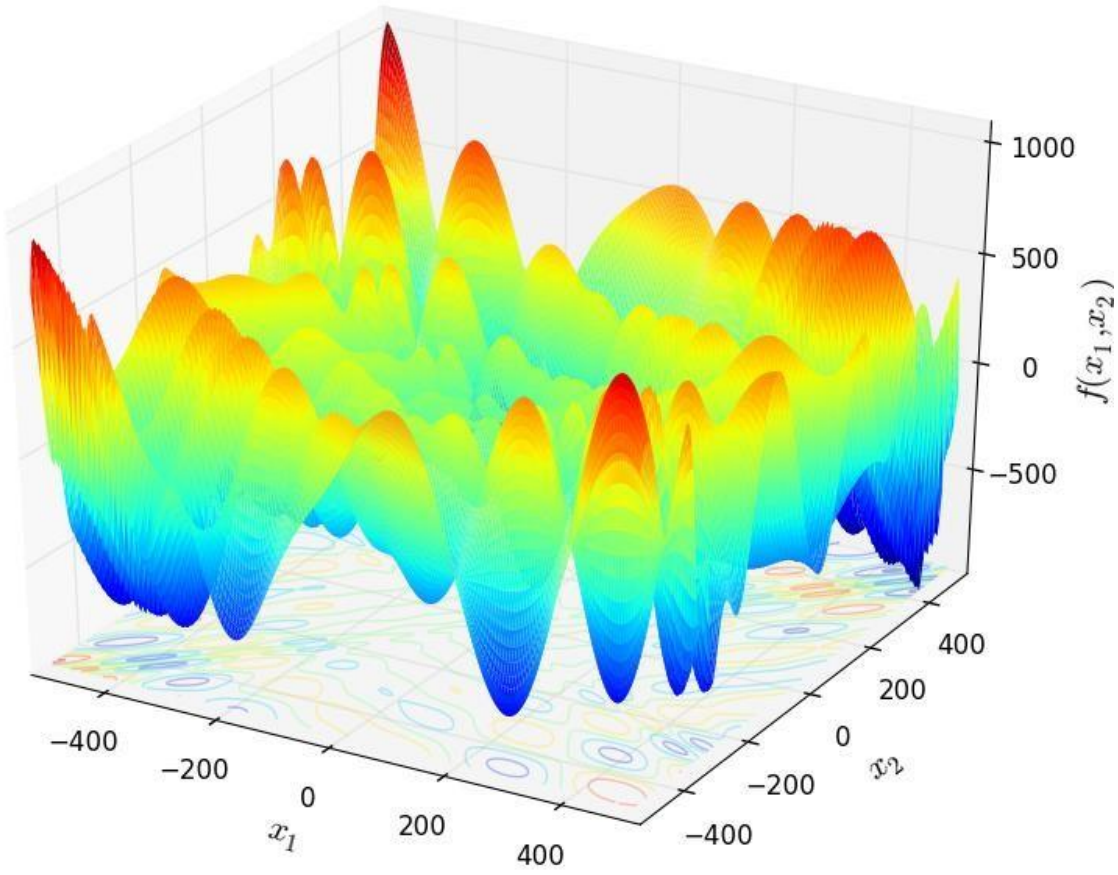
```python
    @staticmethod
    def creep_mutation(offspring):
        print('before mutation', offspring.x, offspring.y, offspring.fitness)
        if random.random() < 0.5:
            if random.random() < 0.5:
                offspring.x = min(2, offspring.x + 0.025)
            else:
                offspring.x = max(-2, offspring.x - 0.025)
        else:
            if random.random() < 0.5:
                offspring.y = min(2, offspring.y + 0.025)
            else:
                offspring.y = max(-2, offspring.y - 0.025)
        offspring.fitness = offspring.evaluate()
        print('after mutation', offspring.x, offspring.y, offspring.fitness)

# Initialize population
pop = [Chromosome(-2, 2, -2, 2) for _ in range(10)]

# Evolutionary Algorithm
for iteration in range(1000):
    new_population = []
    for _ in range(5):
        for _ in range(2):
            parent1 = Selection.binary_tournament(pop, 2)
            parent2 = Selection.binary_tournament(pop, 2)
            offspring1, offspring2 = Operators.simple_crossover(parent1, parent2)
            if random.random() < 0.2:
                if random.random() < 0.5:
                    Operators.creep_mutation(offspring1)
                else:
                    Operators.creep_mutation(offspring2)
            new_population.append(offspring1)
            new_population.append(offspring2)

    combined_pool = pop + new_population
    pop = [Selection.roulette_wheel(combined_pool) for _ in range(10)]

pop.sort(key=lambda item: item.fitness)
best_solution = pop[-1]
print(best_solution.x, best_solution.y, best_solution.fitness)
```

```
before mutation -1.7060721404630432 -1.0434266954512808 1570.8205013824124
after mutation -1.7310721404630431 -1.0434266954512808 1639.649015535045
before mutation 1.6562821172481645 1.8752509049375141 75.77649961110421
after mutation 1.6312821172481646 1.8752509049375141 62.15146533003616
before mutation -1.7060721404630432 -1.1043543875509862 1619.3746649830332
after mutation -1.6810721404630433 -1.1043543875509862 1551.9594928213828
before mutation 1.6562821172481645 -1.1043543875509862 1480.8523967458032
after mutation 1.6562821172481645 -1.1293543875509862 1500.1530209431385
before mutation -1.7060721404630432 -1.1043543875509862 1619.3746649830332
after mutation -1.7310721404630431 -1.1043543875509862 1689.2502654779173
before mutation -1.7060721404630432 -1.1043543875509862 1619.3746649830332
after mutation -1.7060721404630432 -1.1293543875509862 1639.5123476631081
before mutation -1.7060721404630432 -1.1293543875509862 1639.5123476631081
after mutation -1.7310721404630431 -1.1293543875509862 1709.8175911931091
before mutation -1.7060721404630432 -1.1043543875509862 1619.3746649830332
after mutation -1.7310721404630431 -1.1043543875509862 1689.2502654779173
before mutation 1.6562821172481645 -1.1043543875509862 1480.8523967458032
after mutation 1.6562821172481645 -1.1293543875509862 1500.1530209431385
before mutation 1.6562821172481645 -1.1043543875509862 1480.8523967458032
```

Task 2:

Alter your Evolutionary Algorithm implemented in Task 1 to find the maximum value for the Egg Holder function provided below.

$$f(x,y) = -(y+47) \cdot sin\sqrt{\left|\frac{x}{2}+(y+47)\right|} - x \cdot sin\sqrt{|x-(y+47)|}$$



In [2]:

```python
import random
import copy
from math import sin, sqrt, fabs

class Chromosome:

    def __init__(self, x1, x2, y1, y2):
        self.x = random.uniform(x1, x2)
        self.y = random.uniform(y1, y2)
        self.fitness = self.evaluate()

    def evaluate(self):
        return -(self.y + 47) * sin(sqrt(fabs(self.x / 2 + self.y + 47))) - self.x * sin(sqrt(fabs(self.x - (self.y + 47))))

class Selection:
    @staticmethod
    def binary_tournament(pop, T_size):
        tournament_teams = []
        for i in range(T_size):
            selected = random.choice(pop)
            if selected is not None:
                tournament_teams.append(selected)
        if tournament_teams:
            return max(tournament_teams, key=lambda item: item.fitness)
        else:
            return Chromosome(-512, 512, -512, 512)

    @staticmethod
    def roulette_wheel(population):
        fitness_sum = sum([ind.fitness for ind in population])
        selection_point = random.uniform(0, fitness_sum)
        current_sum = 0
        for individual in population:
            current_sum += individual.fitness
            if current_sum > selection_point:
                return individual

class Operators:

    @staticmethod
    def simple_crossover(parent1, parent2):
        child1 = copy.deepcopy(parent1)
        child2 = copy.deepcopy(parent2)
        child1.y = parent2.y
        child2.y = parent1.y
        child1.fitness = child1.evaluate()
        child2.fitness = child2.evaluate()
        return child1, child2

    @staticmethod
    def creep_mutation(offspring):
        print('before mutation', offspring.x, offspring.y, offspring.fitness)
        if random.random() < 0.5:
            if random.random() < 0.5:
                offspring.x = min(512, offspring.x + 0.025)
            else:
                offspring.x = max(-512, offspring.x - 0.025)
        else:
            if random.random() < 0.5:
                offspring.y = min(512, offspring.y + 0.025)
            else:
                offspring.y = max(-512, offspring.y - 0.025)
        offspring.fitness = offspring.evaluate()
        print('after mutation', offspring.x, offspring.y, offspring.fitness)

# Initialize population within the range [-512, 512] for both x and y
pop = [Chromosome(-512, 512, -512, 512) for _ in range(10)]

# Evolutionary Algorithm
for iteration in range(1000):
    new_population = []
    for _ in range(5):
        for _ in range(2):
            parent1 = Selection.binary_tournament(pop, 2)
            parent2 = Selection.binary_tournament(pop, 2)
            offspring1, offspring2 = Operators.simple_crossover(parent1, parent2)
            if random.random() < 0.2:
                if random.random() < 0.5:
                    Operators.creep_mutation(offspring1)
                else:
                    Operators.creep_mutation(offspring2)
            new_population.append(offspring1)
            new_population.append(offspring2)

    combined_pool = pop + new_population
```

```
    pop = [Selection.roulette_wheel(combined_pool) for _ in range(10)]

pop.sort(key=lambda item: item.fitness)
best_solution = pop[-1]
print(best_solution.x, best_solution.y, best_solution.fitness)
```

```
before mutation 139.58111293595783 412.79340504922277 506.11280424134577
after mutation 139.58111293595783 412.7684050492228 506.0173324838576
before mutation 139.58111293595783 412.7684050492228 506.0173324838576
after mutation 139.55611293595783 412.7684050492228 505.87537622833577
before mutation 139.58111293595783 412.79340504922277 506.11280424134577
after mutation 139.60611293595784 412.79340504922277 506.25468505081227
before mutation 139.58111293595783 412.79340504922277 506.11280424134577
after mutation 139.58111293595783 412.7684050492228 506.0173324838576
before mutation 139.58111293595783 412.79340504922277 506.11280424134577
after mutation 139.60611293595784 412.79340504922277 506.25468505081227
before mutation 139.60611293595784 412.79340504922277 506.25468505081227
after mutation 139.63111293595784 412.79340504922277 506.3965028665263
before mutation 139.60611293595784 412.79340504922277 506.25468505081227
after mutation 139.58111293595783 412.79340504922277 506.11280424134577
before mutation 139.60611293595784 412.79340504922277 506.25468505081227
after mutation 139.63111293595784 412.79340504922277 506.3965028665263
before mutation 139.60611293595784 412.79340504922277 506.25468505081227
after mutation 139.60611293595784 412.7684050492228 506.1592257680878
before mutation 139.60611293595784 412.79340504922277 506.25468505081227
```