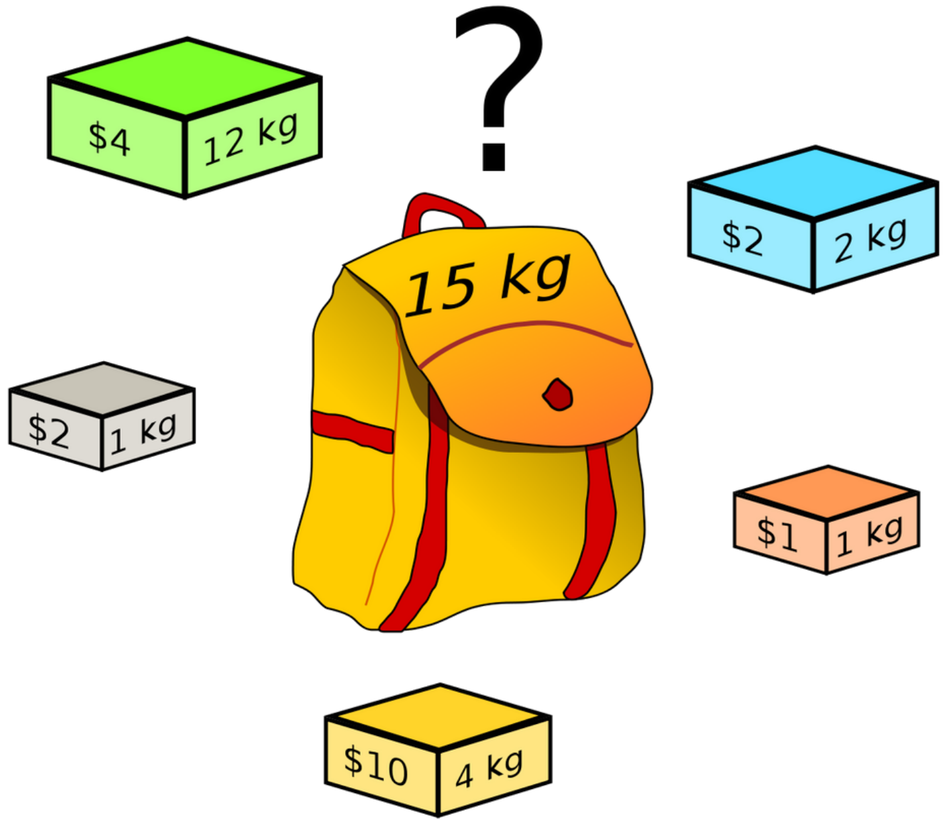
GENETIC ALGORITHM

FOR KNAPSACK PROBLEM

CPE212 ALGORITHM DESIGN

KNAPSACK PROBLEM



WHY? GENETIC ALGORITHM

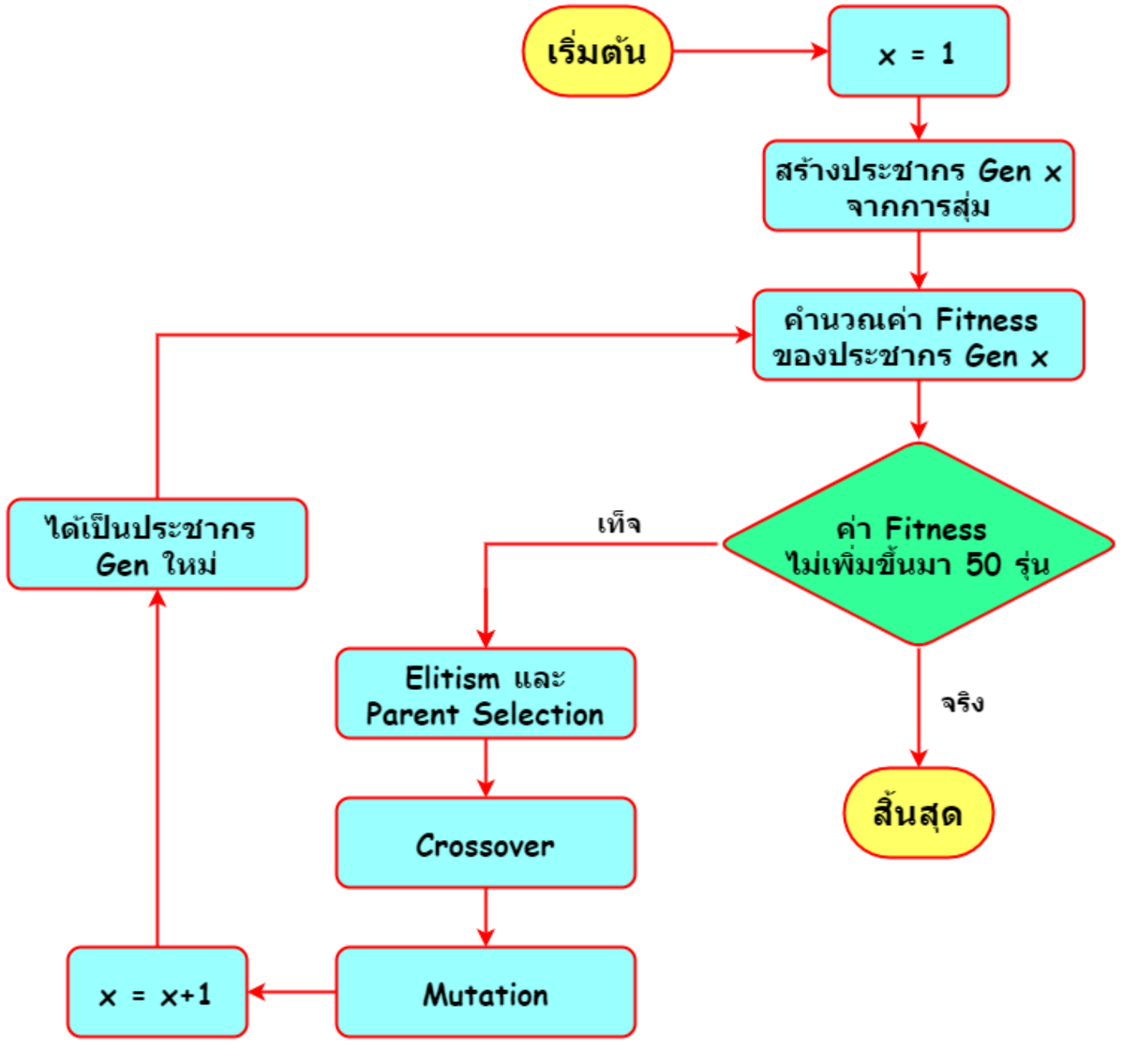
Easy

Nondeter-
ministic

Only
input

GENETIC ALGORITHM OUTLINE

GENETIC ALGORITHM FLOWCHART

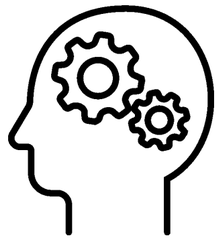


PARAMETERS

- Population Size: 1,000
- Termination Criteria: Fitness 50 generation
- Encoding Process: Binary
- Parent Selection Method: Elitism Selection
- Crossover Method: Uniform
- Crossover Rate: 70%
- Mutation Rate: 40%
- Elitism : 30%
- Number of Runs: 5

CREATING POPULATION

```
[[0 1 1 ... 1 0 1]
 [0 0 0 ... 0 0 0]
 [1 0 0 ... 0 0 1]
 ...
 [0 0 0 ... 0 1 1]
 [0 1 0 ... 1 0 0]
 [0 1 0 ... 1 0 1]]
```



Encoding Process : Binary

Coding

```
import numpy as np
import pandas as pd
import random as rd
import matplotlib.pyplot as plt
import io
import time
from google.colab import files
uploaded = files.upload()
file_name = next(iter(uploaded))
io.StringIO(uploaded[file_name].decode("utf-8"))
item_list=pd.read_csv(io.StringIO(uploaded[file_name].decode("utf-8")))
col = list(item_list.columns)
item_num=int(col[0])
sack_weight=int(col[1])
weight_mean = np.mean(item_list[col[1]])
rate_1 = (sack_weight/weight_mean)/item_num
rate_0 = 1-rate_1

solutions_per_pop = 1000
pop_size = (solutions_per_pop,item_num)
population = np.random.choice([0,1], size = pop_size,p=[rate_0,rate_1])
print(population)
```

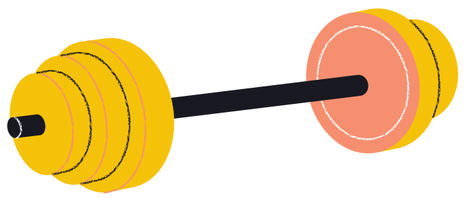
CALCULATING FITNESS

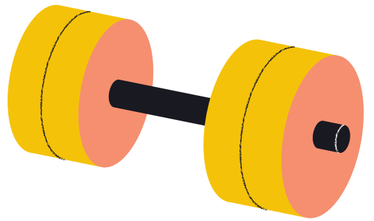
Ex. From Example Set

[0 1 0 0 0 1 0 1 1 1] --> Fitness = 293

[0 1 1 0 1 0 0 1 1 1] --> Fitness = 294

[0 1 1 1 0 0 0 1 1 1] --> Fitness = 295

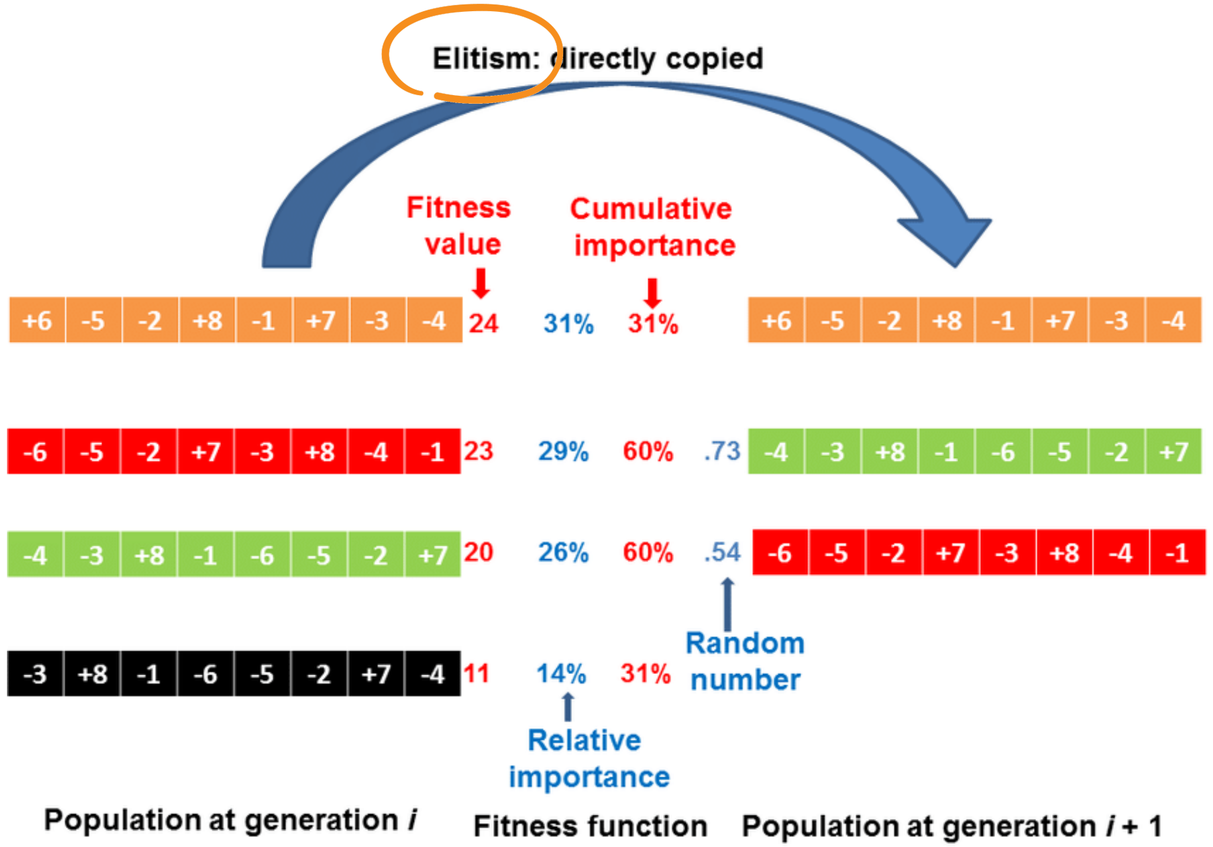




Coding

```
def cal_fitness(weight, value, population, sack_weight):
    fitness = np.empty(population.shape[0])
    for i in range(population.shape[0]):
        sum_value = np.sum(population[i] * value)
        sum_weight = np.sum(population[i] * weight)
        if sum_weight <= sack_weight:
            fitness[i] = sum_value
        else :
            fitness[i] = 0
    return fitness.astype(int)
```

ELITISM & PARENTS SELECTION



Finding population with the best fitness

Coding

```
def elitism_and_selection(fitness, num_elite,num_parents, population):
    fitness = list(fitness)
    elite = np.empty((num_elite, population.shape[1]))
    parents = np.empty((num_parents, population.shape[1]))
    for i in range(num_parents):
        max_fitness_index = np.where(fitness == np.max(fitness))
        parents[i,:] = population[max_fitness_index[0][0], :]
        if(i < num_elite):
          elite[i,:] = population[max_fitness_index[0][0], :]
        fitness[max_fitness_index[0][0]] = -999999
    return elite,parents
```

CROSSOVER

Parent :

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Children :

1 0 0 0 1 1 0 1 0 1 0 0 1 0 0 1 1 1 1 0 1

0 1 1 1 0 0 1 0 1 0 1 1 0 1 1 0 0 0 0 1 0

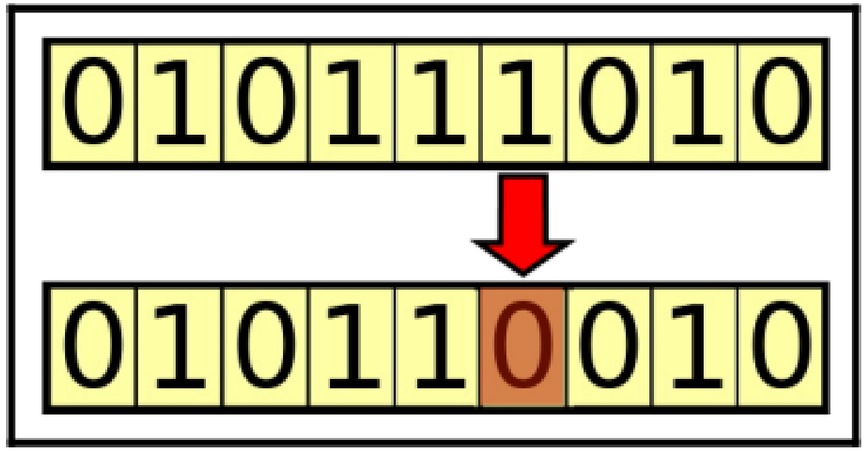
Uniform Crossover

Rate : 70%

Coding

```
def crossover(parents, num_parents):
    offsprings = np.empty((num_parents, parents.shape[1]))
    crossover_rate = 0.7
    i=0
    while (i < num_parents):
        x = rd.random()
        if x > crossover_rate:
          continue
        P = np.random.rand(parents.shape[1])
        parent1_index = i%parents.shape[0]
        parent2_index = (i+1)%parents.shape[0]
        for j in range(parents.shape[1]):
          if(P[j] < 0.5):
            offsprings[i,j] = parents[parent2_index,j]
            offsprings[i+1,j] = parents[parent1_index,j]
          else:
            offsprings[i,j] = parents[parent1_index,j]
            offsprings[i+1,j] = parents[parent2_index,j]
        i+=2
    return offsprings
```

MUTATION



Mutation operator applied to a binary-coded chromosome

Rate : 40%

Coding

```
def mutation(offsprings):
    mutants = np.empty((offsprings.shape))
    mutation_rate = 0.4
    for i in range(mutants.shape[0]):
        mutation_prob = rd.random()
        mutants[i,:] = offsprings[i,:]
        if mutation_prob > mutation_rate:
            continue
        mutation_point = rd.randint(0,offsprings.shape[1]-1)
        if mutants[i,mutation_point] == 0:
            mutants[i,mutation_point] = 1
        else:
            mutants[i,mutation_point] = 0
    return mutants
```

Coding

```
def optimize(weight, value, population, pop_size, sack_weight):
    fitness_history = []
    num_elite = int(pop_size[0]*0.3)
    num_parents = pop_size[0] - num_elite
    termination = 0
    max_fitness = 0
    num_generations = 1
    while termination < 50:
      fitness = cal_fitness(weight, value, population, sack_weight)
      fitness_history.append(fitness)
      if max_fitness < np.max(fitness):
        max_fitness = np.max(fitness)
        termination = 0
      else:
        termination += 1
        if(termination == 50):
          break
      elite,parents = elitism_and_selection(fitness, num_elite,num_parents,population)
      offsprings = crossover(parents, num_parents)
      mutants = mutation(offsprings)
      population[0:num_elite, :] = elite
      population[num_elite:, :] = mutants
      num_generations += 1
    print("Max Fitness: {}\n".format(max_fitness))

    fitness_last_gen = cal_fitness(weight, value, population, sack_weight)
    best_population = np.where(fitness_last_gen == np.max(fitness_last_gen))
    best_chromosome = []
    best_chromosome.append(population[best_population[0][0],:])
    return best_chromosome, fitness_history, num_generations
```

TEST CASE

Set I

1:

Max Fitness: 28599

Number Of Generations: 136

Run Time: 1 minutes 50.29088020324707 seconds

2:

Max Fitness: 28586

Number Of Generations: 148

Run Time: 1 minutes 59.705941677093506 seconds

3:

Max Fitness: 28663

Number Of Generations: 151

Run Time: 2 minutes 2.8963723182678223 seconds

4:

Max Fitness: 28603

Number Of Generations: 156

Run Time: 2 minutes 6.558988094329834 seconds

5:

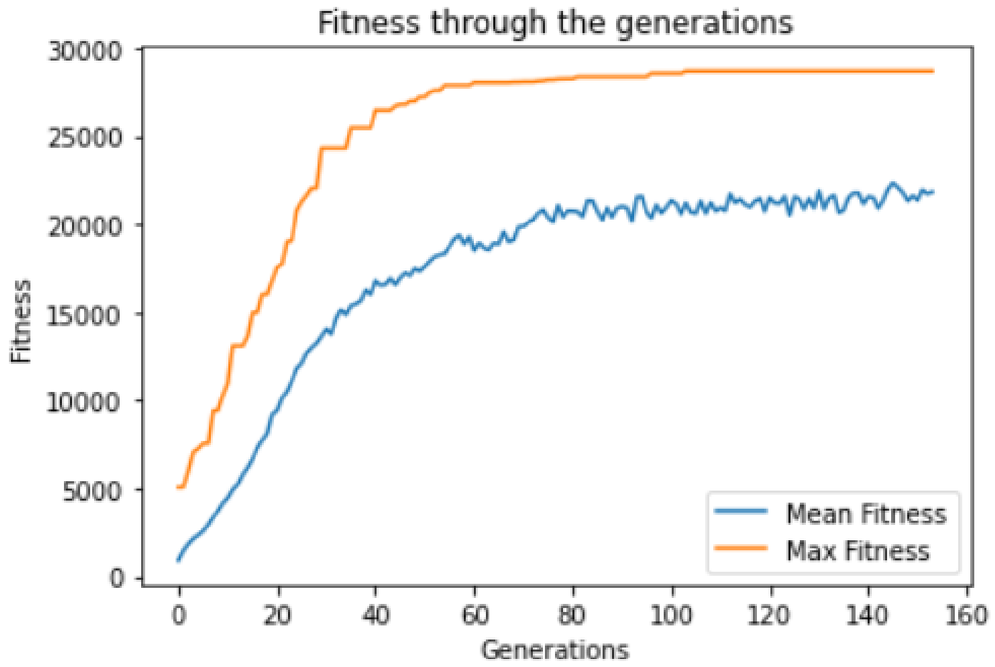
Max Fitness: 28703

Number Of Generations: 154

Run Time: 2 minutes 5.374474287033081 seconds

TEST CASE

The best of Set I



Max Fitness: 28703

Number Of Generations: 154

Run Time: 2 minutes 5.374474287033081 seconds

Selected items:

7 11 13 14 24 26 33 36 38 39 49 54 61 122 135 138 147
148 216 217 237 246 250 255 270 274 282 335 348 363
374 380 383 420 422 427 447 464 470 474 477 494 495

TEST CASE

Set II

1:

Max Fitness: 53516

Number Of Generations: 192

Run Time: 3 minutes 36.84612584114075 seconds

2:

Max Fitness: 54148

Number Of Generations: 191

Run Time: 3 minutes 33.888145446777344 seconds

3:

Max Fitness: 53053

Number Of Generations: 199

Run Time: 3 minutes 45.3716094493866 seconds

4:

Max Fitness: 53778
Number Of Generations: 281

Run Time: 4 minutes 37.88637709617615 seconds

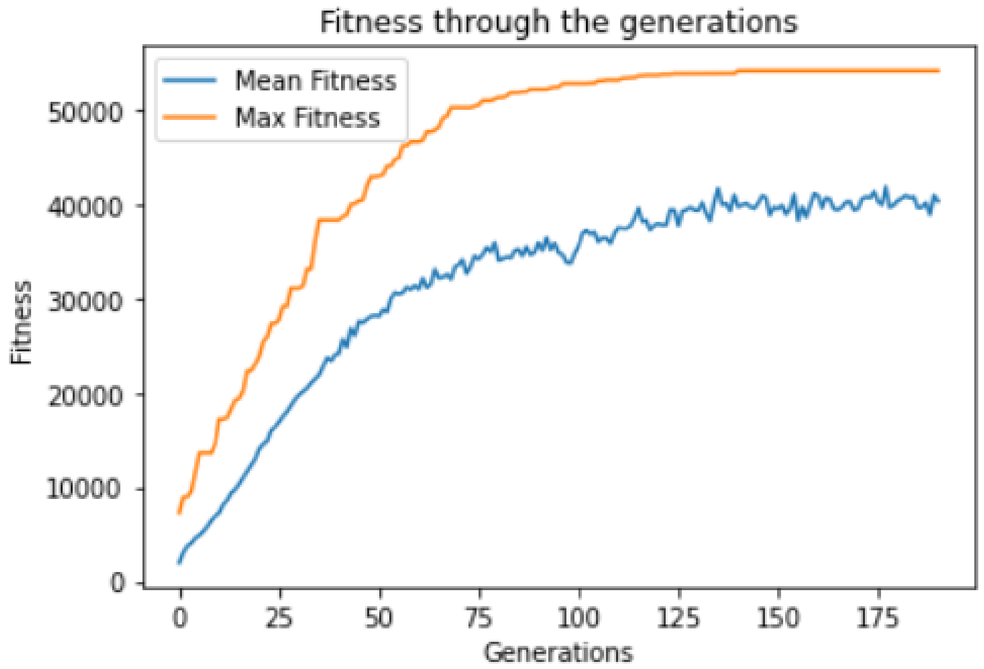
5:

Max Fitness: 52098
Number Of Generations: 208

Run Time: 3 minutes 25.95441460609436 seconds

TEST CASE

The best Set II



Max Fitness: 54148

Number Of Generations: 191

Run Time: 3 minutes 33.888145446777344 seconds

Selected items:

7 11 14 24 26 33 36 38 39 49 54 61 122 135 138 147
148 152 216 217 237 250 255 263 274 282 335 348 363
363 374 380 383 420 422 427 447 470 474 477 481 494
495 574 586 593 600 604 611 613

TEST CASE

Set III

1:

Max Fitness: 107551
Number Of Generations: 387

Run Time: 7 minutes 45.07540321350098 seconds

2:

Max Fitness: 105851
Number Of Generations: 366

Run Time: 7 minutes 18.038255214691162 seconds

3:

Max Fitness: 106675
Number Of Generations: 353

Run Time: 7 minutes 21.173062324523926 seconds

4:

Max Fitness: 107696
Number Of Generations: 485

Run Time: 9 minutes 39.450101375579834 seconds

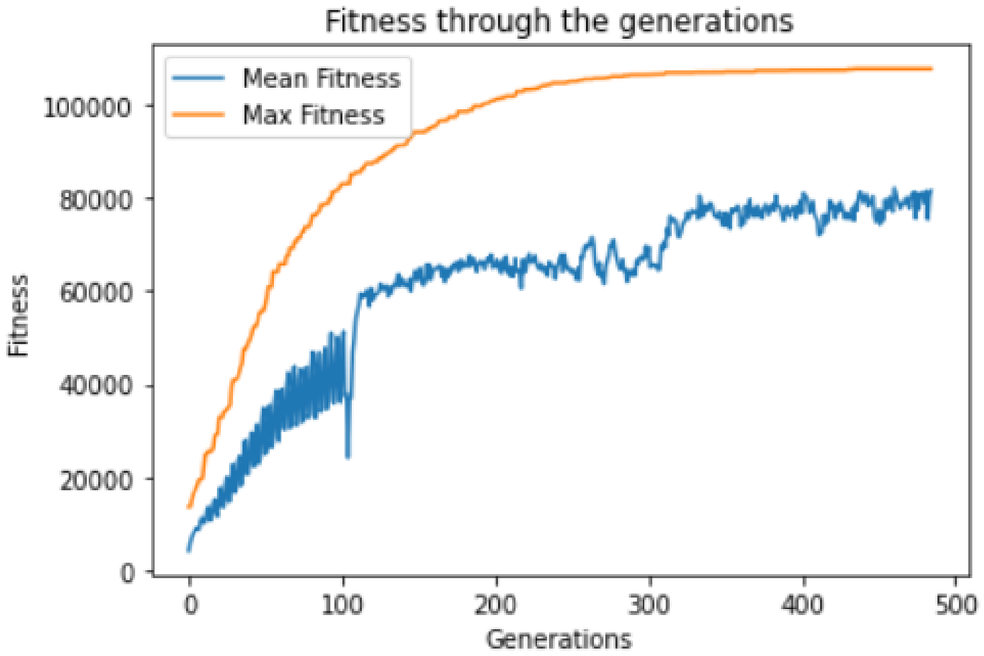
5:

Max Fitness: 103206
Number Of Generations: 298

Run Time: 5 minutes 53.120327949523926 seconds

TEST CASE

The best of Set III



Max Fitness: 107696
Number Of Generations: 485

Run Time: 9 minutes 39.450101375579834 seconds

Selected items:
7 11 13 24 33 38 39 49 54 61 122 135 147 148 217 250
255 270 274 282 335 348 363 374 380 383 420 422 427
447 470 474 477 481 494 495 574 593 600 604 611 613
658 704 709 719 733 737

CONS OF GENETIC ALGORITHM

Slow

Not Optimal

Parameters

GENETIC ALGORITHM VS. DYNAMIC PROGRAMMING

หัวข้อการเปรียบเทียบ (The Best)	Genetic Algorithm			Dynamic Programming		
	Set 1	Set 2	Set 3	Set 1	Set 2	Set 3
Max Fitness	28703	54148	107696	28857	54503	110625
Run Time	2.05 m	3.34 m	9.39 m	0.24 s	1.12 s	8.91 s

หัวข้อในการเปรียบเทียบ	Genetic Algorithm	Dynamic Programming
เวลาในการ run	ใช้เวลาในการ run มากกว่า	ใช้เวลาในการ run น้อยกว่า
ผลลัพธ์ของการ run	ดีตามเวลาและรุ่นที่เพิ่มขึ้น	ดีที่สุด
หน่วยความจำที่ใช้	ขึ้นอยู่กับจำนวนประชากร	มาก
การนำไปใช้	ง่ายต่อการประยุกต์ใช้ แต่การกำหนด parameters ให้เหมาะสมนั้นยาก	ง่ายต่อการประยุกต์ใช้
ความยากในการใช้งาน	ไม่จำเป็นต้องใช้สมการในการคำนวน ใช้เพียงแค่ input เท่านั้น	ต้องตีโจทย์ออกมาเป็นสมการทางคณิตศาสตร์เพื่อใช้ในการคำนวน

OUR PROBLEM

เอกสารอ้างอิง

https://en.wikipedia.org/wiki/Genetic_algorithm

https://www.educative.io/edpresso/what-is-the-knapsack-problem

https://stackoverflow.com/questions/9146086/time-complexity-of-genetic-algorithm

https://medium.com/koderunners/genetic-algorithm-part-3-knapsack-problem-b59035ddd1d6

https://medium.com/@samiranbera_66038/crossover-operator-the-heart-of-genetic-algorithm-6c0fdcb405c0

https://th.wikipedia.org/wiki/ปัญหาถุงกระสอบ

https://medium.com/@aquablitz11/an-introduction-to-dynamic-programming-76574fda6501

OUR MEMBERS

KUNANON	SUPMAMUL	63070501011
THANADOL	THONGRIT	63070501029
TANASEAD	RATTANAPAN	63070501033
SAKCHAI	PAOIN	63070501059

THANK YOU FOR YOUR ATTENTION