FINAL REPORT

GENETIC ALGORITHM

FOR KNAPSACK PROBLEM

CPE212
ALGORITHM DESIGN

PRESENT

DR.NATASHA DEJDUMRONG

BY

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

คำนำ

รายงานฉบับนี้เป็นส่วนหนึ่งของวิชา Algorithm Design (CPE212) ของนักศึกษาชั้นปีที่ 2 ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ โดยมีจุดประสงค์เพื่อนำ Genetic Algorithm มาประยุกต์ใช้กับ Knapsack Problem เพื่อให้เกิดประสิทธิภาพในการทำงานที่ดีที่สุด

ทั้งนี้ เนื้อหาได้รวบรวมมาจากหนังสือแบบเรียนและแหล่งข้อมูลจากอินเทอร์เน็ต ขอขอบคุณ รศ.ดร. ณัฐชา เดชดำรง อย่างสูงที่กรุณาตรวจ และขอขอบคุณเพื่อน ๆ ภายในภาควิชาที่ให้คำแนะนำและข้อเสนอตลอดการทำงาน

คณะผู้จัดทำหวังว่ารายงานฉบับนี้จะเป็นประโยชน์ต่อผู่อ่านไม่มากก็น้อย

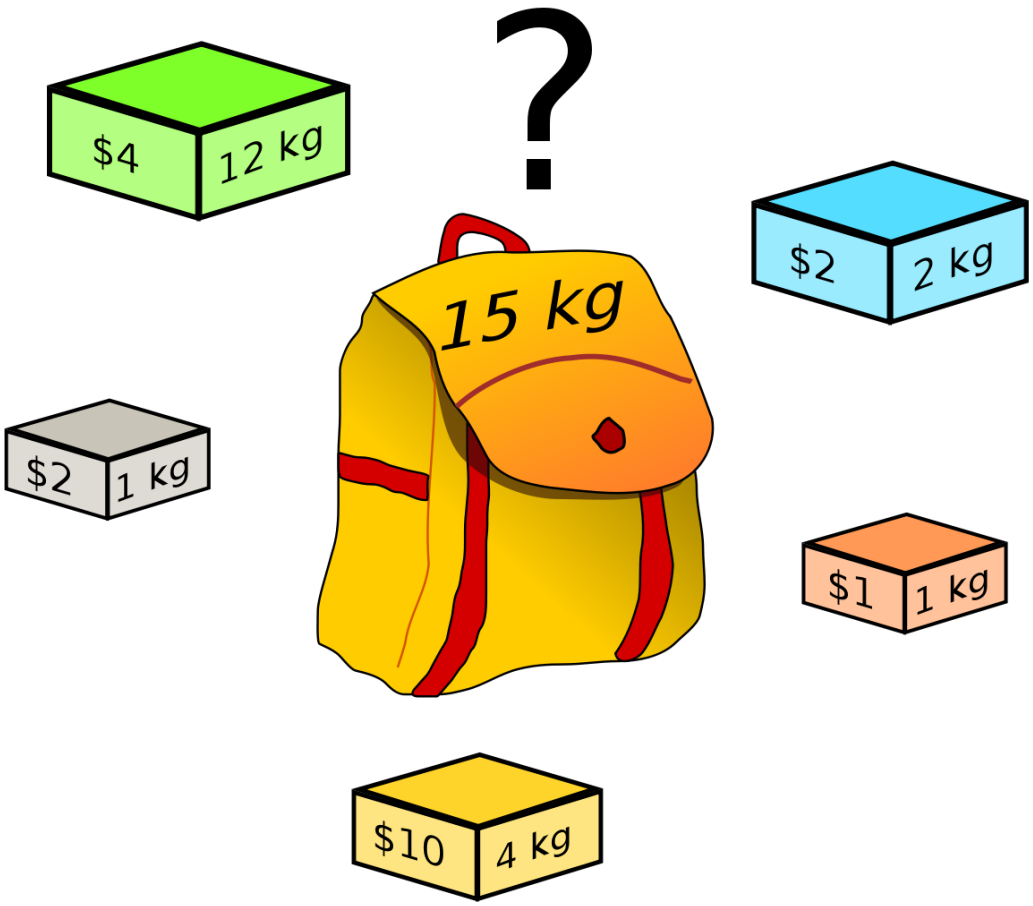
คณะผู้จัดทำ

สารบัญ

Knapsack Problem คืออะไร?

Knapsack Problem คือปัญหาการเขียนโปรแกรมแบบไดนามิกที่มีชื่อเสียงซึ่งจัดอยู่ในหมวดการเพิ่มประสิทธิภาพ

ได้ชื่อมาจากสถานการณ์สมมติ การเลือกหยิบของใส่ถุงโดยให้มีมูลค่ารวมสูงที่สุด แต่น้ำหนักโดยรวมต้องไม่เกินน้ำหนักที่บรรจุได้ โดยของแต่ละชิ้นจะมีน้ำหนักและมูลค่าแตกต่างกันไป โดยแต่ละรายการสามารถเลือกได้เพียงครั้งเดียว

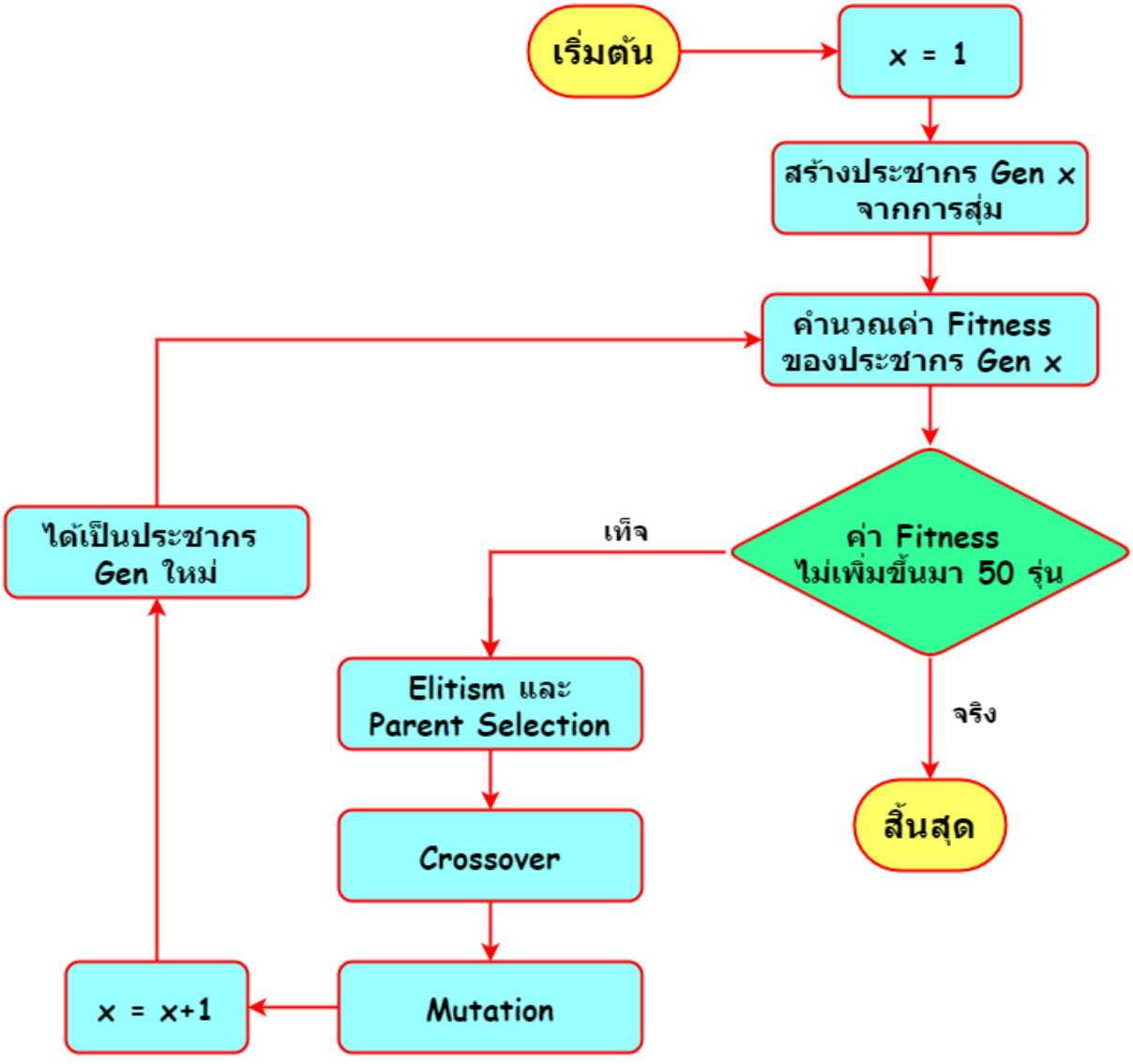


Genetic Algorithm (GA) คืออะไร?

Genetic Algorithm คืออัลกอริทึมที่เป็น กระบวนการค้นหาโดยจะเลือกวิธีการแก้ปัญหาที่เหมาะสมที่สุด (metaheuristic) โดยมีแรงบัลดาลใจมาจากกระบวนการการคัดเลือกโดยธรรมชาติ ที่ซึ่งเป็นการวิวัฒนาการของสิ่งมีชีวิตกลุ่มใหญ่

โดยทั่วไป Genetic Algorithm มักใช้เพื่อเพิ่มความถูกต้องของการแก้ปัญหา โดยการเพิ่มประสิทธิภาพและค้นหาปัญหา โดยอาศัยวิธีการที่ได้รับแรงบัลดาลใจมาจากวิธีการทางชีวภาพของสิ่งมีชีวิต เช่น การกลายพันธุ์ ครอสโอเวอร์ และการคัดเลือก เป็นต้น

Flow Chart (ผังงาน)



Pseudocode (โค้ดเทียม)

```
termination <- 0
max_fitness <- 0
x <- 1
Create Random Population of Gen x
While termination < 50
    Calculate Fitness of Gen x
    If max_fitness < Fitness of Gen x
        max_fitness <- Fitness of Gen x
        termination <- 0
    Else
        termination <- termination + 1
        If termination == 50
            End While
    elite,parents <- Do elitism_and_selection(Gen x)
    offsprings <- Do crossover(parents)
    mutants <- Do mutation(offsprings)
    x <- x+1
    Gen x = elite + mutants
```

- 1: ให้ termination = 0, max_fitness = 0, x = 1
- 2: สร้างประชากรแบบสุ่มของ Gen x
- 3: ในขณะที่ termination < 50 ทำข้อ 4-16
- 4: คำนวณ Fitness ของ Gen x
- 5: ถ้า max_fitness < Fitness ของ Gen x ทำข้อ 6,7
- 6: ให้ max_fitness = Fitness ของ Gen x
- 7: ให้ termination = 0
- 8: ถ้าไม่เข้าเงื่อนไขในข้อ 5 ทำข้อ 9-11
- 9: ให้ค่าของ termination เพิ่มขึ้น 1
- 10: ถ้า terminination = 50
- 11: จบ while loop
- 12: elite,parents = elitism_and_selection(Gen x)
- 13: offsprings = crossover(parents)
- 14: mutants = mutation(offsprings)
- 15: ให้ค่าของ x เพิ่มขึ้น 1
- 16: ให้ Gen x = elite + mutants

Parameters

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

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]]
```

ประชากร(Population) คือกลุ่มของวิธีการแก้ปัญหา
ข้อมูลแต่ละแถวเปรียบเสมือนประชากรแต่ละตัว โดยเลขแต่ละหลักเปรียบเสมือนยีนในประชากรนั้น ๆ
(ยีน 0 คือไม่ได้เลือกของชิ้นนั้นใส่ถุง ยีน 1 คือเลือกของชิ้นนั้นใส่ถุง)

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)
```

1: โค้ดในส่วนของการนำเข้าข้อมูลด้วยการ import file การกำหนดตัวแปรต่าง ๆ และการสร้างประชากรรุ่นที่ 1

```
[ ]  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)
```

2: โค้ดในส่วนของการคำนวนค่า fitness
(fitness คือค่าของมูลค่า(value) เมื่อแก้ปัญหาด้วยประชากรตัวนั้น ๆ)

```
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
```

3: โค้ดในส่วนของการคัดเลือกประชากร elite และการเลือก parents โดยใช้วิธีการเลือกจากประชากรที่มีค่า fitness มากที่สุด

```
[ ] 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
```

4: โค้ดในส่วนของการทำ uniform crossover ด้วยการสุ่มค่าความน่าจะเป็นในตำแหน่งต่าง ๆ ของยีนแต่ละคู่ ถ้าความน่าจะเป็นในตำแหน่งนั้น ๆ น้อยกว่า 0.5 ให้สลับยีนที่ตำแหน่งนั้น (เสมือนการโยนเหรียญหัวก้อยว่าตำแหน่งไหนจะได้ crossover กัน)

```
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
```

5: โค้ดในส่วนของการทำ mutation ด้วยการสลับค่ายีนหนึ่งยีน (จากเลข 0 เป็น 1 จากเลข 1 เป็น 0) ภายในประชากรแต่ละตัว โดยมีการสุ่มตำแหน่งที่จะเกิดการ mutation

```
[ ] fitness_history_mean = [np.mean(fitness) for fitness in fitness_history]
    fitness_history_max = [np.max(fitness) for fitness in fitness_history]
    plt.plot(list(range(num_generations)), fitness_history_mean, label = 'Mean Fitness')
    plt.plot(list(range(num_generations)), fitness_history_max, label = 'Max Fitness')
    plt.legend()
    plt.title('Fitness through the generations')
    plt.xlabel('Generations')
    plt.ylabel('Fitness')
    plt.show()
    print(np.asarray(fitness_history).shape)
```

6: โค้ดในส่วนของการแสดงผลของการ run ในรูปแบบ graph

```
[ ]  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
```

7: โค้ดในส่วนของการทำ Reproduction ประชากร และการคำนวนหลัก ๆ ของ Genetic Algorithm

```
[ ]  start = time.time()
     best_chromosome, fitness_history, num_generations = optimize(item_list[col[1]], item_list[col[0]], population, pop_size, sack_weight)
     end = time.time()
     total_time = end-start
     minutes = int(total_time/60)
     seconds = total_time - (minutes*60)
     print("Number Of Generations: {}\n ".format(num_generations))
     print("Run Time: {} minutes {} seconds\n".format(minutes,seconds))
     selected_items = item_num * best_chromosome
     print('\nSelected items:')
     for i in range(len(selected_items)):
       if selected_items[0][i] != 0:
         print('{}'.format(i+1),end = " ")
```

8: โค้ดสั่งการและ โค้ดในส่วนของการแสดงผลการ run

กรณีทดสอบ (Test Case)

Set: 1 Max weight: 2,543 Number of items: 500

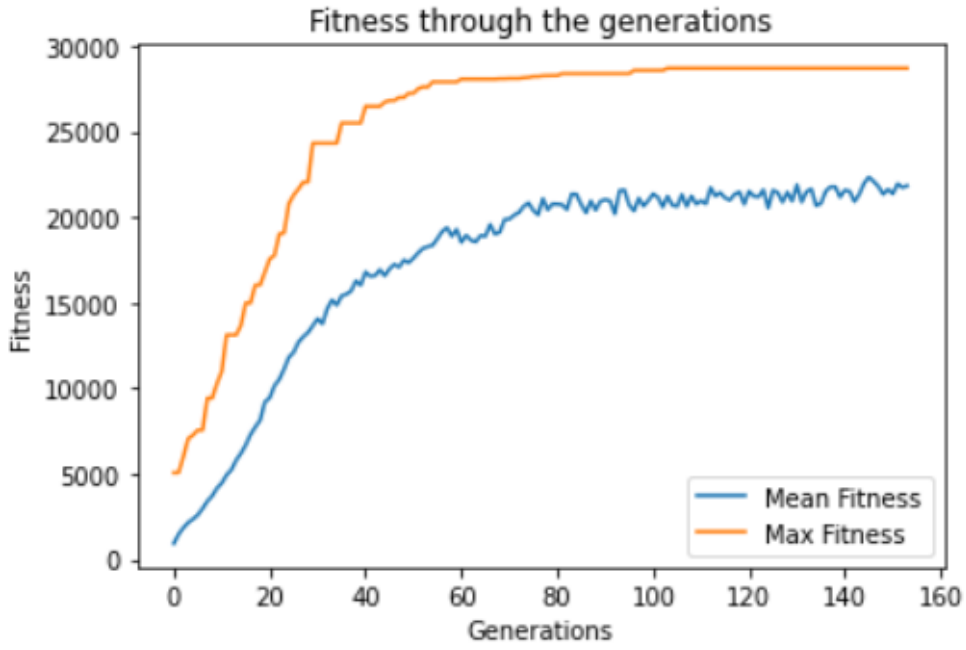
ผลลัพธ์ที่ดีที่สุดจากการ run ทั้งหมด 5 ครั้ง วัดจากค่า Max Fitness

```
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
```



Set: 2 Max weight: 5,002 Number of items: 1,000

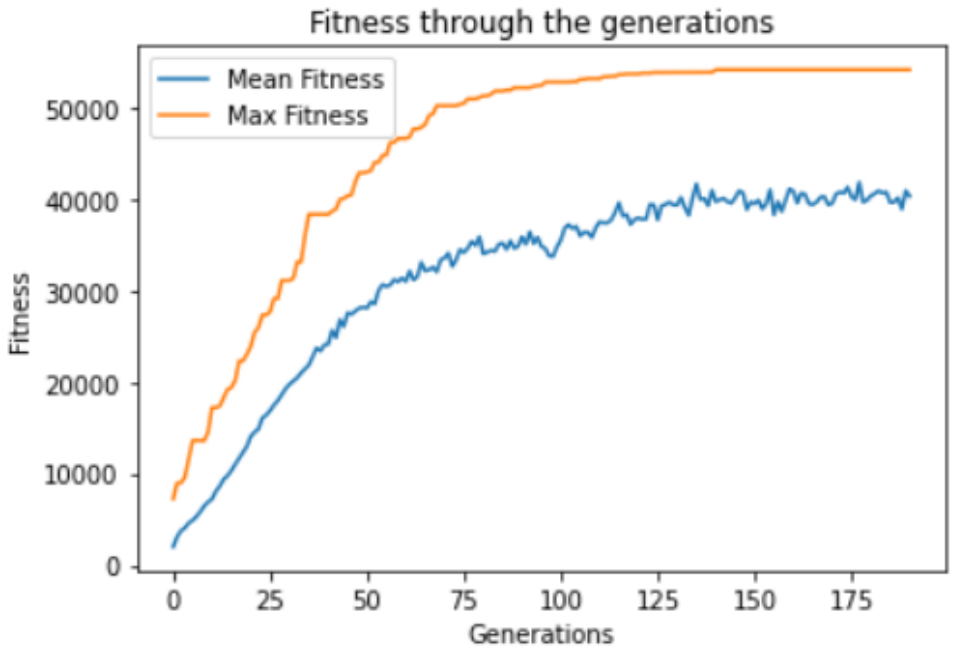
ผลลัพธ์ที่ดีที่สุดจากการ run ทั้งหมด 5 ครั้ง วัดจากค่า Max Fitness

```
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
217 237 250 255 263 274 282 335 348 363 374 380 383 420 422
427 447 470 474 477 481 494 495 574 586 593 600 604 611 613
```

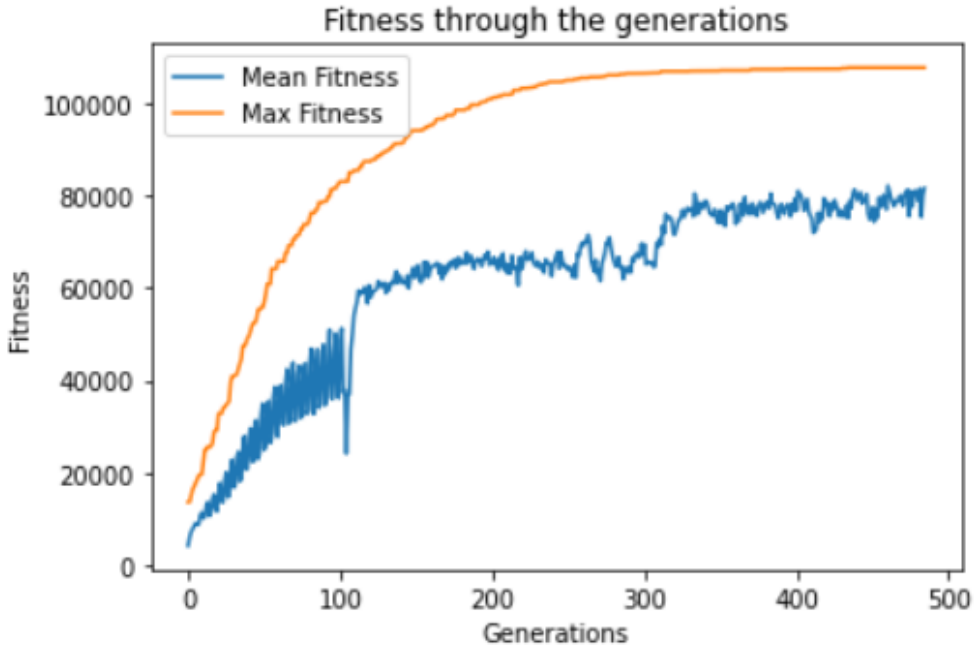


Set: 3 Max weight: 10,011 Number of items: 2,000
ผลลัพธ์ที่ดีที่สุดจากการ run ทั้งหมด 5 ครั้ง วัดจากค่า Max Fitness

```
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
495 574 593 600 604 611 613 658 704 709 719 733 737 738 740 744
752 771 776 787 822 823 825 831 837 846 856 887 915 938 946 968
987 988 990 1004 1035 1043 1044 1049 1058 1097 1100 1115 1118
1138 1139 1144 1159 1217 1240 1247 1249 1261 1273 1299 1323 1334
1344 1345 1354 1362 1395 1401 1420 1432 1434 1458 1464 1467 1473
1474 1476 1484 1486 1498 1508 1533 1534 1545 1546 1548 1549 1550
1568 1592 1605 1616 1618 1631 1639 1655 1665 1667 1673 1691 1693
1717 1725 1730 1733 1825 1862 1865 1874 1883 1891 1901 1905 1916 1954
1955 1998
```

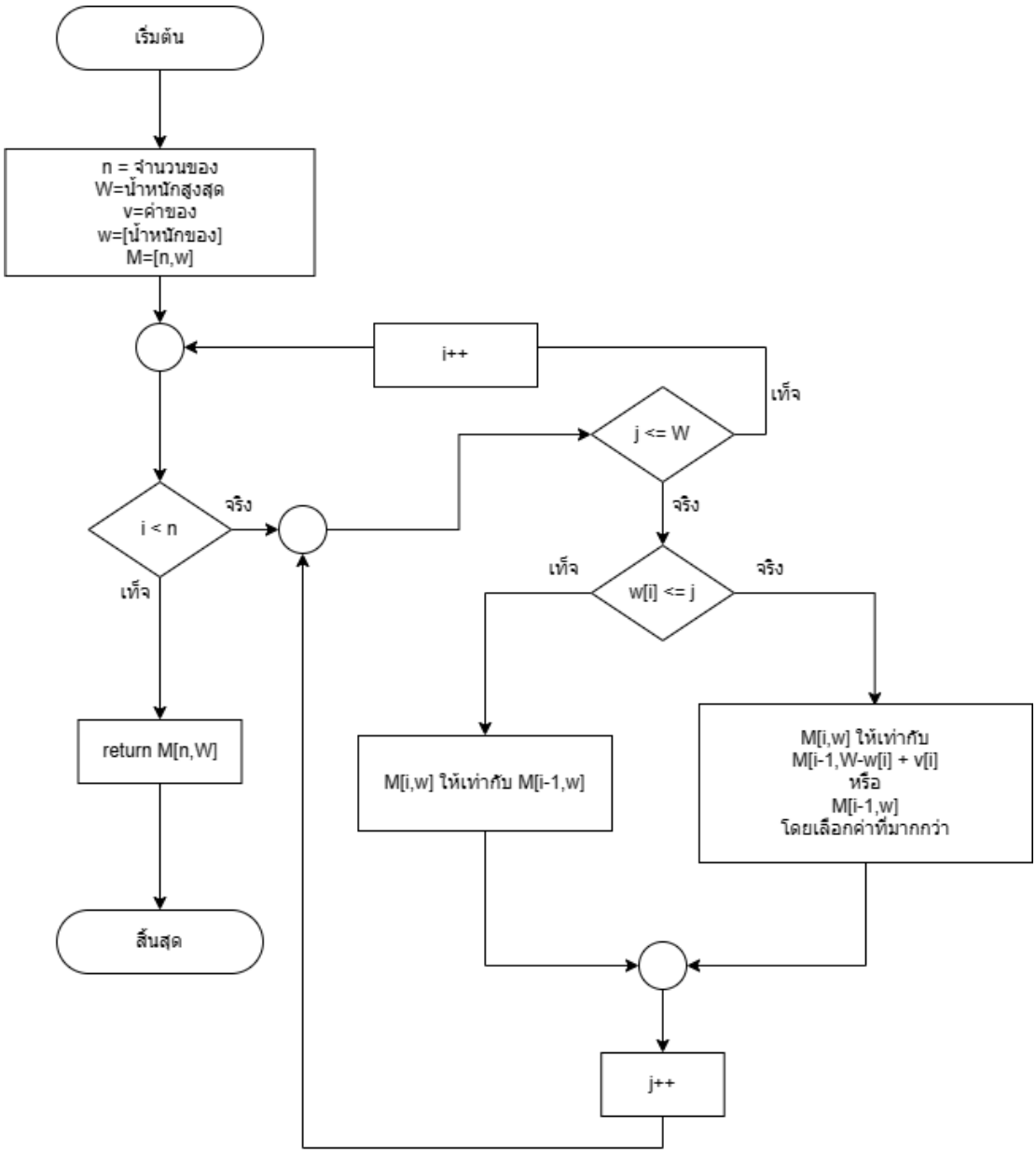


Dynamic Programing คืออะไร?

Dynamic Programing เป็นเทคนิคหนึ่งสำหรับการแก้ปัญหาที่ซับซ้อน โดยเริ่มจากการแก้ปัญหาย่อย (ปัญหาที่มีขนาดเล็กลงมา) ตั้งแต่ปัญหาขนาดย่อยที่สุดขึ้นมาก่อน แล้วค่อยๆเพิ่มขอบเขตขึ้นมาจนถึงปัญหาที่ใหญ่ที่สุด

เนื่องจากตัว ปัญหาย่อย (Subproblem) มีลักษณะการแก้ปัญหาที่เหมือนกัน แค่มีข้อมูลให้จัดการน้อยกว่า ดังนั้นวิธีการแก้ปัญหา DP มักจะต้องแก้ด้วย recursive function หรือไม่ก็ต้องใช้สูตรคณิตศาสตร์ ที่มีลักษณะเป็นสมการเวียนเกิด (Recurrence Formula)

Flow Chart (ผังงาน)



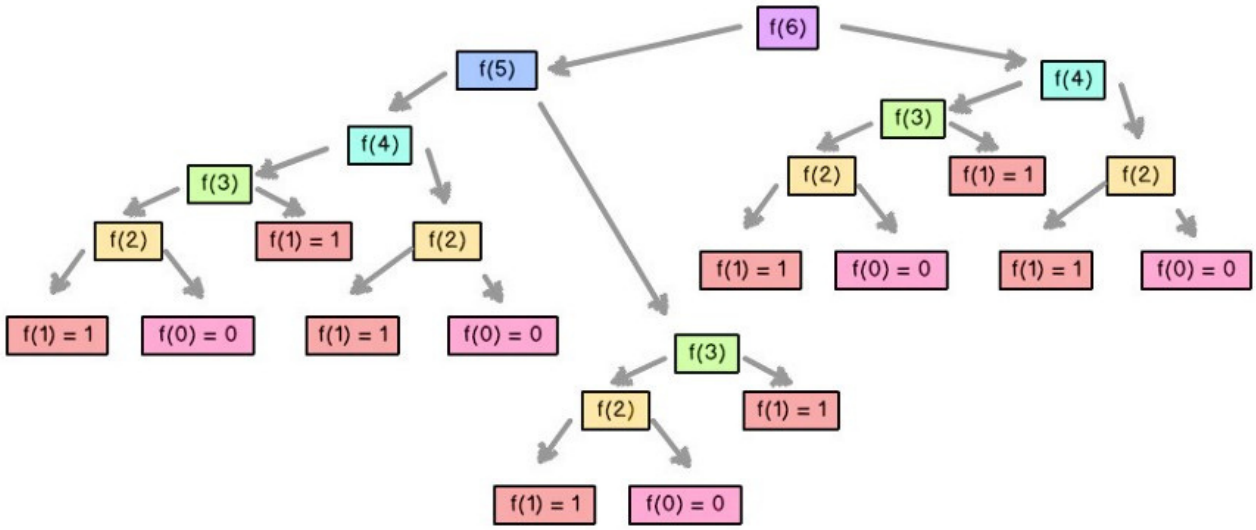
Pseudocode (โค้ดเทียม)

Algorithm 1: Dynamic Programming Algorithm for 0-1 Knapsack Problem

Data: $W, v_1, v_2, v_3, ..., v_n\ ,\ w_1, w_2, w_3, ..., w_n$
Result: $M[n, W]$
$M[0, w] \leftarrow 0,\ \forall w\ 0\ to\ W$
$M[i, 0] \leftarrow 0,\ \forall i\ 0\ to\ n$
for $i \leftarrow 1\ to\ n$ **do**
  for $w \leftarrow 1\ to\ W$ **do**
    if $w_i \leq w$ **then**
      $M[i, w] = max(M[i-1, w-w_i]\ +\ v_i,\ M[i-1, w])$
    else
      $M[i, w] = M[i-1, w]$
    end
  end
end
$return\ M[n, W]$

- 1: ข้อมูล weight & volume
- 2: ผลลัพธ์ M[n, W] (Array 2D)
- 3: ให้ M[0, w] = 0 ; w ตั้งแต่ 0 ถึง W
- 4: ให้ M[i, 0] = 0 ; i ตั้งแต่ 0 ถึง n
- 5: for loop i = 1 ถึง n
- 6: for loop w = 1 ถึง W
- 7: ถ้า w(i) <= w ให้
- 8: ให้ M[i, w] = max(M[i - 1, w - w(i) + v(i), M[i - 1, w])
- 9: ถ้าไม่ใช่เงื่อนไขก่อนหน้า
- 10: ให้ M[i, w] = M[i - 1, w]
- 11: end if else
- 12: end for loop
- 13: end for loop
- 14: แสดงผลค่า M[n, w]

ตัวอย่างการทำงาน



Coding

```
async function main(){
    const n = await read("Data set : ");
    const data = readFile(n);
    const ans = knapsack(data);
    console.log('\nMax Fitness : ' + ans['Max Fitness']);
    console.log('\nSelected item : ' + ans['Selected item']);
    console.log('\nRun Time : ' + ans['Run Time'] + ' seconds');
}

main();
```

1: โค้ดในส่วนของฟังก์ชัน main()

```
function max(a, b) {
    return (a > b) ? [a, 0] : [b, 1];
}
```

2: โค้ดในส่วนของฟังก์ชัน max หาค่าที่มากระหว่างค่า 2 ค่า

```
function knapsack(data) {
    let i, w, temp, ok;
    let k = [];
    let s = [];
    let t = Date.now();

    for (i = 0; i <= data.count; i++) {
        for (w = data.max; w >= 0; w--) {

            if (i==0) {
                k[w] = 0;
                s[w] = [];
            }
            else if (data.list[i-1].weight <= w) {
                temp = w - data.list[i-1].weight;
                [k[w], ok] = max(data.list[i-1].value + k[temp], k[w]);

                if (ok == true)
                    s[w] = [...new Set([...s[temp], ...[i]])];
            }
        }
    }
    t = Date.now() - t;
    return {
        'Fitness': k[data.max],
        'Time': t / 1000,
        'Select':s[data.max]
    }
}
```

3: โค้ดในส่วนของฟังก์ชันการคำนวนและแก้ปัญหา knapsack

```
import _readline from 'readline';
import fs from 'fs'
import path from 'path';

const __dirname = path.resolve();

const readline = _readline.createInterface({
    input: process.stdin,
    output: process.stdout
});
```

4: โค้ดในส่วนของการ import library ในการอ่านไฟล์ และรับ input

```
function read(str){
    return new Promise((resolve, rejects) => {
        readline.question(str, (input) => {
            readline.close();
            resolve(input);
        })
    })
}
```

```
function readFile(n){
    let str = fs.readFileSync(path.join(__dirname, `Set${n}.txt`), 'utf8');
    let count = 0;
    let max = 0;
    let list = [];
    str.split('\n').forEach((item, i) => {
        const data = item.split(' ');
        if (i == 0) {
            count = parseInt(data[0]);
            max = parseInt(data[1].replace('\r', ''));
        } else {
            list.push({
                'value': parseInt(data[0]),
                'weight': parseInt(data[1].replace('\r', ''))
            });
            wt.push(parseInt(data[1].replace('\r', '')));
            val.push(parseInt(data[0]));
        }
    });
    return { 'count': count, 'max': max, 'list': list}
}
```

5: โค้ดในส่วนของฟังก์ชันการรับ input และการอ่านไฟล์

กรณีทดสอบ (Test Case)

Set: 1 Max weight: 2,543 Number of items: 500

ผลลัพธ์ที่ดีที่สุดจากการ run ทั้งหมด 5 ครั้ง วัดจากค่า Max Fitness

```
Data set : 1

Max Fitness : 28857

Selected item : 7 11 13 14 24 26 33 38 39 49 54 61 122 135 138 147 148
148 152 216 217 250 255 270 274 282 335 348 363 374 380 383 420 422 427
447 464 470 474 477 481 494 495
  Run Time : 0.238 seconds
```

Set: 2 Max weight: 5,200 Number of items: 1,000

ผลลัพธ์ที่ดีที่สุดจากการ run ทั้งหมด 5 ครั้ง วัดจากค่า Max Fitness

```
Data set : 2

Max Fitness : 54503

Selected item : 7 11 13 14 24 26 33 36 38 39 49 54 61 122 135 138 147 148
152 217 237 246 250 255 270 274 282 335 348 363 374 380 383 420 422 427
447 464 470 474 477 481 494 495 540 574 586 593 599 600 604 611 613 658
658 670 704 709 719 733 737 738 744 752 771 776 787 823 825 831 846 850 856
887 888 915 938 946 968 985 987 988 990 993
  Run Time : 1.199 seconds
```

Set: 3 Max weight: 10,011 Number of items: 2,000

ผลลัพธ์ที่ดีที่สุดจากการ run ทั้งหมด 5 ครั้ง วัดจากค่า Max Fitness

```
Data set : 3

Max Fitness : 110625

Selected item : 7 11 14 24 26 33 38 39 49 54 61 122 135 138 147 148 152 217
147 148 152 217 237 246 250 255 270 274 282 335 348 363 374 380 383 420 422
427 447 464 470 474 477 481 494 495 540 574 586 593 600 604 611 613 658 670
704 709 719 733 737 738 744 752 771 776 787 823 825 831 846 850 856 887 888
915 938 946 968 987 988 990 993 1004 1032 1035 1044 1049 1058 1097 1100 1104
1115 1118 1138 1139 1144 1159 1169 1240 1247 1249 1261 1273 1299 1323 1334
1344 1345 1354 1362 1395 1401 1404 1420 1432 1442 1458 1464 1467
1473 1474 1476 1484 1486 1508 1533 1545 1546 1549 1550 1568 1592 1605 1616
1618 1631 1639 1655 1665 1667 1668 1673 1691 1693 1697 1717 1725 1730 1732
1732 1826 1862 1865 1874 1883 1891 1901 1905 1916 1920 1954 1955 1990 1998
  Run Time : 8.91 seconds
```

ข้อดี - ข้อเสียของ Genetic Algorithm

ข้อดีของ Genetic Algorithm คือสามารถใช้เพื่อหาคำตอบของโจทย์ทุกประเภทโดยไม่จำเป็นต้องรู้สมการเราก็สามารถหาคำตอบของสมการหรือระบบนั้นได้เลย ขอแค่เรามี input ก็เพียงพอ
ส่วนข้อเสียของ Genetic Algorithm คือ การทำงานจะช้า และคำตอบที่ได้จากการทำงานของ Algorithm ตัวนี้ในการรันแต่ละครั้งจะได้คำตอบออกมาไม่เหมือนกันและอาจจะไม่ใช่คำตอบที่ดีที่สุด

การเปรียบเทียบ (Comparison)

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	ต้องดีโจทย์ออกมาเป็นสมการทางคณิตศาสตร์เพื่อใช้ในการคำนวน

สรุปผล (Conclusion)

สรุปโดยรวมอย่างง่ายคือ Dynamic Programming มีประสิทธิภาพในการแก้ปัญหา Knapsack Problem ได้ดีกว่า Genetic Algorithm ทั้งในเรื่องของผลลัพธ์และเวลา แต่ Genetic Algorithm จะมีข้อได้เปรียบเรื่องของหน่วยความจำที่หยืดหยุ่นได้ตามจำนวนประชากร และความง่ายในการนำไปใช้งาน

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