AI - Report

Genetic algorithyms – Knapsack Problem

Author: Alejandro Manuel Gestoso Torres
Professor: Agustín Riscos Núñez
Subject: Artificial Inteligence

Index

Introduction 2

Implementation of the algorithms 3

Representations 3

Modifications for the multi-knapsack 5

Instances considered 7

Statistics 8

Knapsack 1 8

Knapsack 2 9

Knapsack 3 10

Multi-Knapsack 1 11

Multi-Knapsack 2 11

Multi-Knapsack 3 12

Bibliography 13

Introduction

The main goal of this assignment consists of develop a genetic algorithm, using python, to solve the knapsack problem and its variant, the multi-size knapsack problem.

The multi-size knapsack states that:

Given a list of items L, where each item has a weight associated with it, the problem is to find a partition of the items into several subsets associated with multiple knapsacks, in such a way that the free space in the knapsacks is minimized.
We will assume that we can use a finite number of sizes for the knapsacks (the list of allowed sizes/capacities should be provided as input).
We will assume that we can use an arbitrary number of knapsacks of the same size.

In this report I will write about how the standard basic algorithm was made and how was modified to fulfill the multi-size algorithm. After that, I will show the graphics about how effective the algorithm was, what was the best solution found and the statistics.

The weights used for the algorithm are the prepared for the Practice 4 (Genetic Algorithms). The weights are these:

```
# _______________________________________________________
# Knapsack problem 1:
# 10 objects, maximum weight 165
weights1 = [23,31,29,44,53,38,63,85,89,82]



# _______________________________________________________
# Knapsack problem 2:
# 15 objects, maximum weight 750

weights2 = [70,73,77,80,82,87,90,94,98,106,110,113,115,118,120]



# _______________________________________________________
# Knapsack problem 3:
# 24 objects, maximum weight 6404180
weights3 = [382745,799601,909247,729069,467902, 44328, 34610,698150,823460,
       903959,853665,551830,610856, 670702,488960,951111,323046,446298,931161,
       31385,496951,264724,224916,169684]
```

Implementation of the algorithms

Representations

To represent this problem, we must define the individuals of the population:

- Genes: Whether the item is on the knapsack or not, it is a value between [0,1]. In case of being in the knapsack, it will have a value of a 1, and in case of not being in it, it will have a 0.

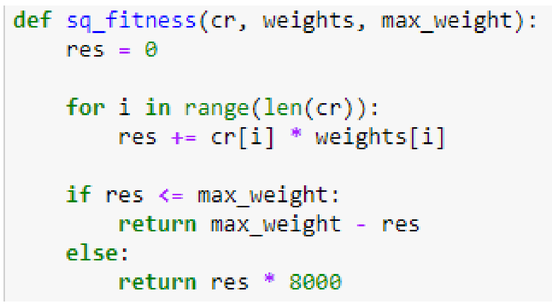
- Positions: A list of weights of every item. For example, given a list of weights like [50,60,70], it means that the item with index 0 will have weight 50, the item with index 1 will have weight 60 and the item with index 2 will have weight 70.

- Population: Every possible solution, that must be evaluated with the fitness function to whether discard (or not) it, depending on how much value has.

And we must define the functions:

- Decode: It must be defined a function that receives a chromosome and interprets its data returning the possible solution, so the fitness function can evaluate it.

- Fitness: This function is used to evaluate how good is the answer proposed by the chromosome and giving a value which will be used afterwards in the tournament selection to pick the best chromosome. The function used is this one:



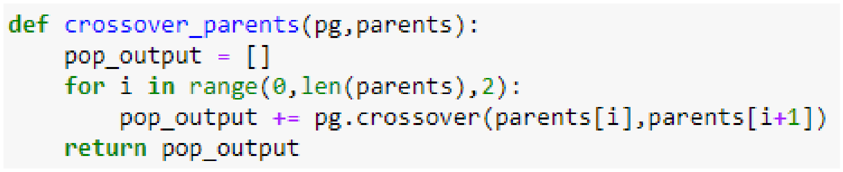
```python
def sq_fitness(cr, weights, max_weight):
    res = 0

    for i in range(len(cr)):
        res += cr[i] * weights[i]

    if res <= max_weight:
        return max_weight - res
    else:
        return res * 8000
```

As we can see, in this function, for every chromosome we obtain its weight and if it is bigger than the maximum weight allowed, we impose a penalty by multiplying its weight by 8000, this make that the answers that are not correct has a value of fitness so bigger that in the future tournament selection, they will always lose.

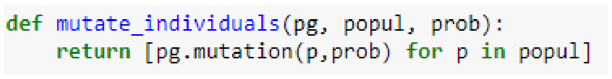
- Crossover: This function receives an instance of the problem and a population of parents and breeds them to obtain new population to continue the experimentation. The function used is this one:



```python
def crossover_parents(pg,parents):
    pop_output = []
    for i in range(0,len(parents),2):
        pop_output += pg.crossover(parents[i],parents[i+1])
    return pop_output
```

In this case, this function is given us by the teacher and ensures that given a list of parents, they breed a new population of children.

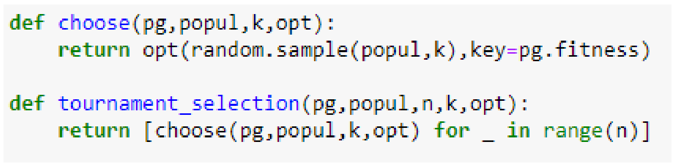
- Mutation: This function receives a population and a probability of mutation and whether changes or not the population according to that given probability. The used function is this one:



```
def mutate_individuals(pg, popul, prob):
    return [pg.mutation(p,prob) for p in popul]
```

This function is given us by the teacher too.

- Tournament selection: This function receives a population, two numbers and condition and picks the number of better possible solutions considering its fitness and the condition, that can be maximum (max) or minimum (min) fitness. The function used is:

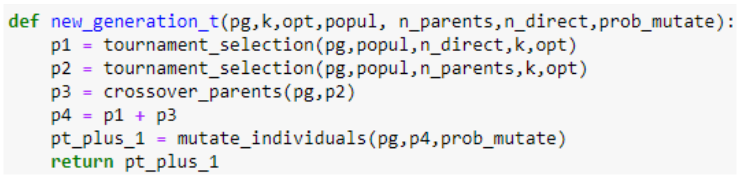


```
def choose(pg,popul,k,opt):
    return opt(random.sample(popul,k),key=pg.fitness)

def tournament_selection(pg,popul,n,k,opt):
    return [choose(pg,popul,k,opt) for _ in range(n)]
```

In this case, the function was given by the teacher and about the two number I talked about upwards, the first one (n) is the number of winners of the tournament and the second one (k) is the number of best picks chosen between the population. It also has an auxiliary function named choose that picks the best possible solution that after that, will be selected as winners in the main function.

- New generation: This function uses all the functions described upwards combined to obtain a new complete generation. The function used is:



```
def new_generation_t(pg,k,opt,popul, n_parents,n_direct,prob_mutate):
    p1 = tournament_selection(pg,popul,n_direct,k,opt)
    p2 = tournament_selection(pg,popul,n_parents,k,opt)
    p3 = crossover_parents(pg,p2)
    p4 = p1 + p3
    pt_plus_1 = mutate_individuals(pg,p4,prob_mutate)
    return pt_plus_1
```

This function was given by the teacher and as we can see, is a combination of the previous ones, because, it makes a tournament selection, a crossover and a mutation and returns a brand-new generation.

- Genetic algorithm: This function uses all the previous functions to provide a solution to the problem.

```
def genetic_algorithm_t(pg,k,opt,ngen,size,ratio_cross,prob_mutate):
    p0 = initial_population(pg,size)
    threshold = round(size * ratio_cross)
    if threshold % 2 == 0:
        n_parents = threshold
    else:
        n_parents = threshold - 1
    n_direct = size - n_parents

    for _ in range(ngen):
        p0 = new_generation_t(pg,k,opt,p0, n_parents,n_direct,prob_mutate)

    res = opt(p0, key = pg.fitness)
    return str(pg.decode(res)) + ',' + str(pg.fitness(res))
```

The function uses the populator to generate an initial population, calculates a threshold and ensures that is a number divisible by two, that will make that the number of parents always will be an even number, and if not, by subtracting one to the number, will be even. Then we run the new generation algorithm and return a possible solution and its fitness.

Modifications for the multi-knapsack

To face the challenge to adapt the genetic algorithm to solve the multi-knapsack problem, many adjustments were considered, for example, creating two types of genes, one to say if the item were selected or not and other to locate to which knapsack was assigned. But the final idea was simply to use the genes to locate where goes every item, having 0 to the items that are not in any knapsack, and use the numbers 1 to x to express the knapsack where the item goes.

The modifications made to adapt the knapsack algorithm to the multi-knapsack are these:

- The decode function: The new decode function has to be more complex than 0,1 values if the item is in or not in the knapsack, so, in order to decode the chromosomes of the answer, we apply this function, having as a result a dictionary where the key is the knapsack that we are talking about and the values are a list of 1s and 0s if the number is in the answer or not.

```
def range_decode(x, size):
    res = {}

    for i in range(size + 1):
        chromosomes = []
        for (j,b) in enumerate(x):
            if x[j] == i:
                chromosomes.append(1)
            else:
                chromosomes.append(0)
        res[i] = sum(b*(2**i) for (i,b) in enumerate(chromosomes))

    return res
```

- The decimal to binary function: To obtain a list of 1s and 0s to represent the number of the chromosomes that we will evaluate in the fitness function.

```python
def decimal_to_binary(x):
    res = []

    binary = '{0:b}'.format(int(x))
    for i in range(len(binary)):
        res.append(int(binary[i]))

    return res
```

- The fitness function: The new fitness function needs an auxiliary one, that evaluates the fitness of every knapsack of the multi-knapsack. After that, the main function sums the fitness of every knapsack to an only number that summarizes the total fitness of the possible solution of the multi-knapsack.

```python
def sq_fitness_multi(cr, weights, max_weight):
    res = 0

    for i in range(len(cr)):
        res += cr[i] * weights[i]

    if res <= max_weight:
        return max_weight - res
    else:
        return res * 8000

def sq_knapsack_fitness(cr, weights, max_weights):
    res = 0

    for i in range(len(max_weights)):
        res += sq_fitness_multi(decimal_to_binary(cr[i + 1]), weights, max_weights[i])

    return res
```

- The main genetic algorithm function: The main algorithm is roughly the same, the only difference is that this function needs the number of knapsacks that we are considering in the problem.

```python
def genetic_algorithm_t_multi(pg,k,opt,ngen,size,ratio_cross,prob_mutate,knapsack_size):
    p0 = initial_population(pg,size)
    threshold = round(size * ratio_cross)
    if threshold % 2 == 0:
        n_parents = threshold
    else:
        n_parents = threshold - 1
    n_direct = size - n_parents

    for _ in range(ngen):
        p0 = new_generation_t(pg,k,opt,p0, n_parents,n_direct,prob_mutate)

    res = opt(p0, key = pg.fitness)
    return str(pg.decode(res,knapsack_size)) + ', ' + str(pg.fitness(res))
```

All the functions that are not described on this section are the same in the knapsack and the multi-knapsack algorithms.

Instances considered

Before writing about the instances considered, I want to talk a bit about the data generation. It is a function where we declare the instances that we want to run the algorithm and the results are written in a CSV file, for further processing and analysis on excel.

This is an example of the data generation function for the knapsack:

```
with open('knapsack1.csv', 'w', newline='') as csvfile:
    reader = csv.writer(csvfile)
    reader.writerow(['chromosome','fitness'])
    for _ in range(instances):
        row = genetic_algorithm_t(knapsack1,3,min,300,8,0.7,0.3).split(',')
        reader.writerow(row)
```

And this is an example of the data generation function for the multi-knapsack:

```
with open('multiKnapsack1.csv', 'w', newline='') as csvfile:
    reader = csv.writer(csvfile)
    reader.writerow(['remaining_obj','ac_weight','knapsack','ac_weight','knapsack','ac_weight','knapsack','ac_weight','total_fitness'])
    for _ in range(instances):
        row = genetic_algorithm_t_multi(multiKnapsack1,3,min,300,8,0.7,0.3,len(max_weights1)).replace('{','').replace('}','').replace(':',',').split(',')
        reader.writerow(row)
```

As we can see, they are roughly the same function, the main difference is because the data generated in the genetic algorithm, the first one gives us the chromosome and the fitness, whereas the second one gives us the knapsack where the item goes and the total weight of the knapsack, and the total fitness of the chromosome.

The instances considered for almost all the algorithms were 300, that ensures us to obtain a good number of results to analyze. That worked fine for every knapsack except of the weights of the knapsack 3, that happens because the weights are so big that every result was unique, so as the results were not conclusive. The solve this problem, the number of instances for this knapsack were 3000, this made that at least some results were repeated, so as we could have some results to analyze.

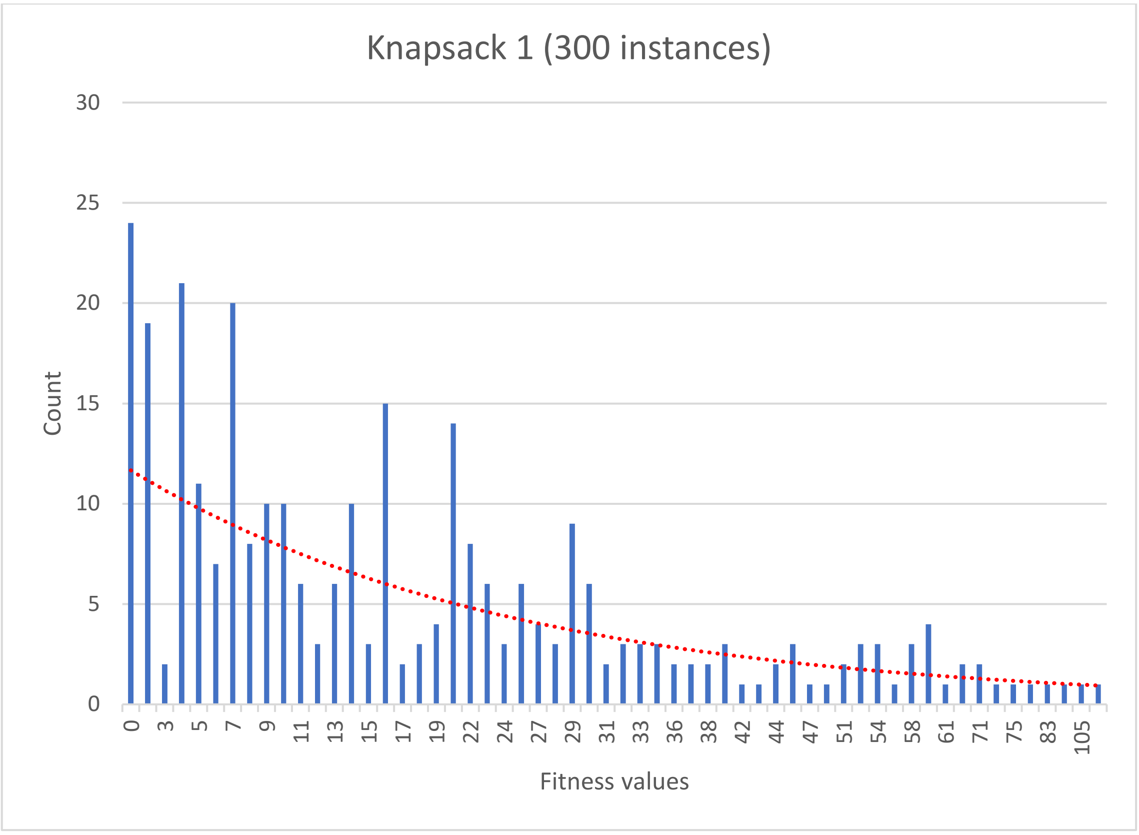
Statistics

To ensure that the data analyzed were accessible for everyone, the data used to make the graphics were made by copying the csv to make them static and generate the graphics using the csv lector provided by Microsoft Excel.

The data considered to make the graphic are the fitness and the number of chromosomes that has the same fitness, and we also provide an exponential tendency line.

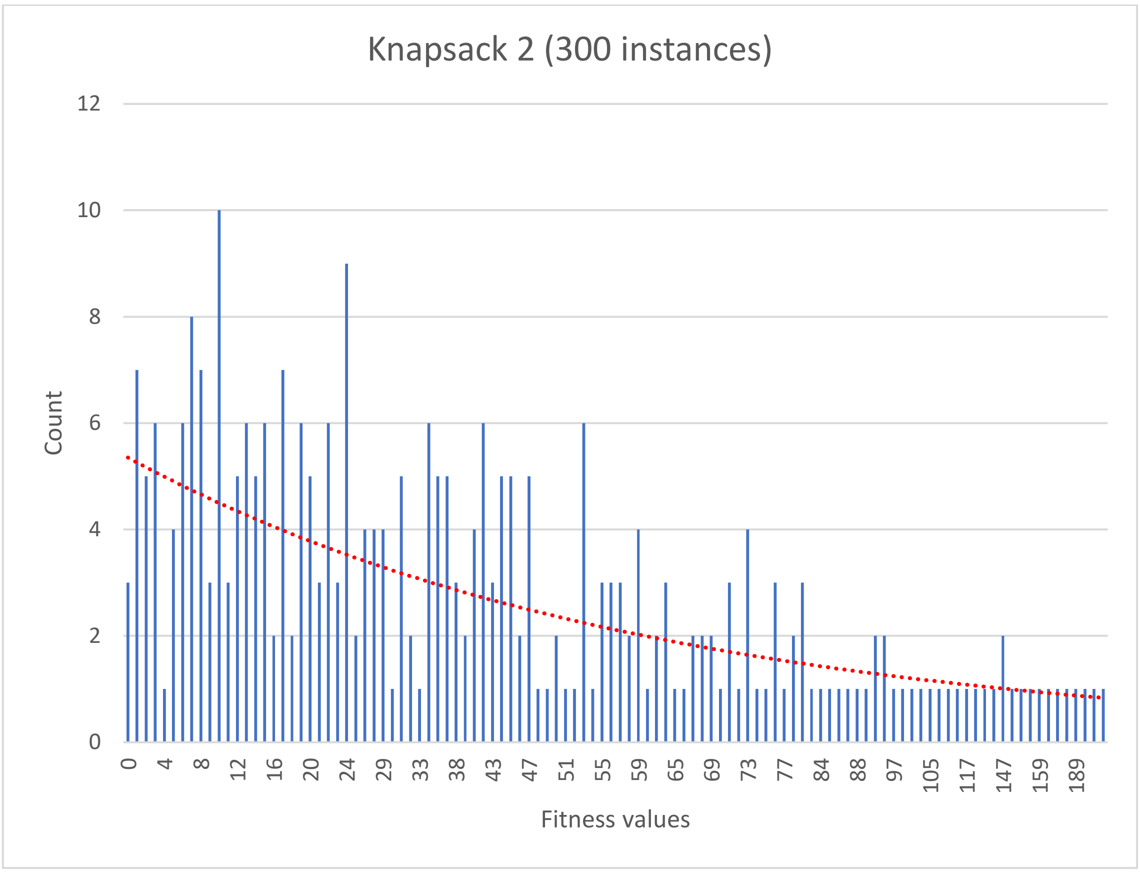
Knapsack 1

In this knapsack, we found 24 different chromosomes with a fitness value of 0, that means that there are 24 perfect solutions. Over the 300 instances, the genetic algorithm found 58 possible solutions.



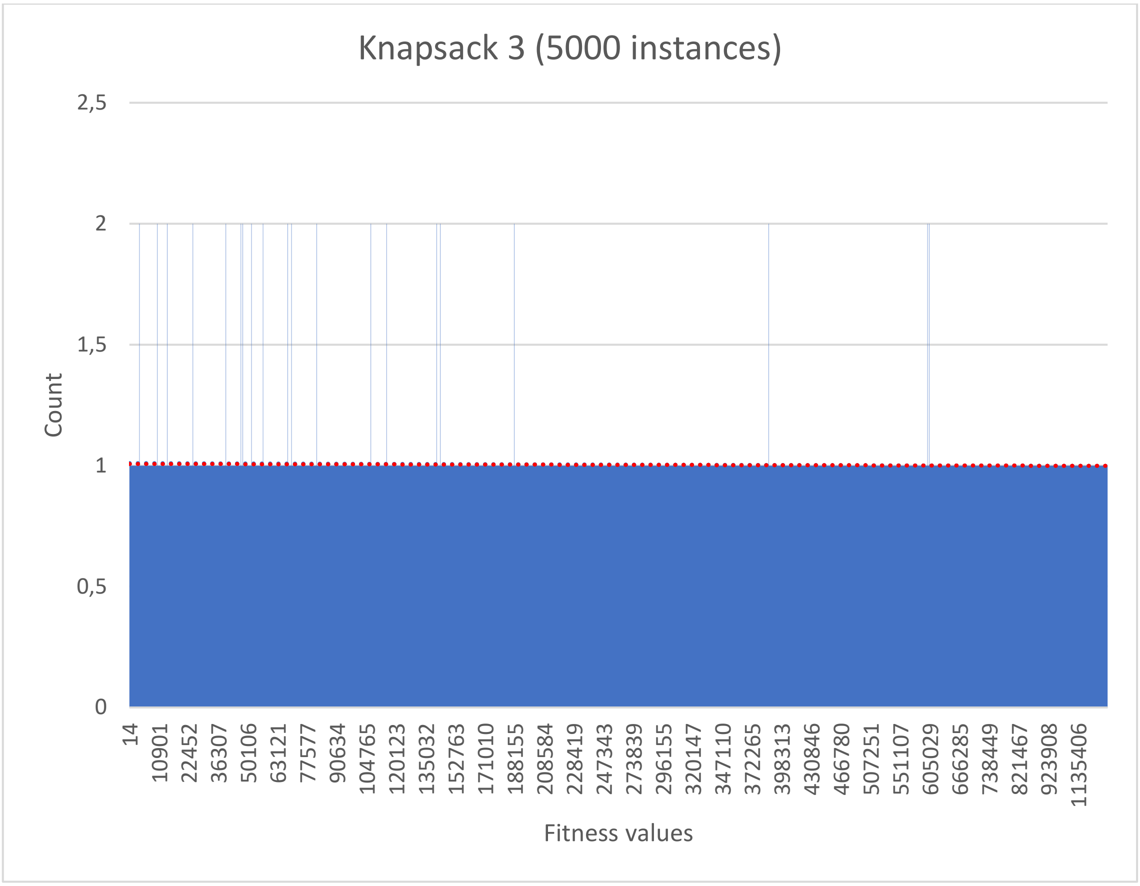
Knapsack 2

In this knapsack, we found 3 different chromosomes with a fitness value of 0, that means that there are 3 perfect solutions. Over the 300 instances, the genetic algorithm found 108 possible solutions.



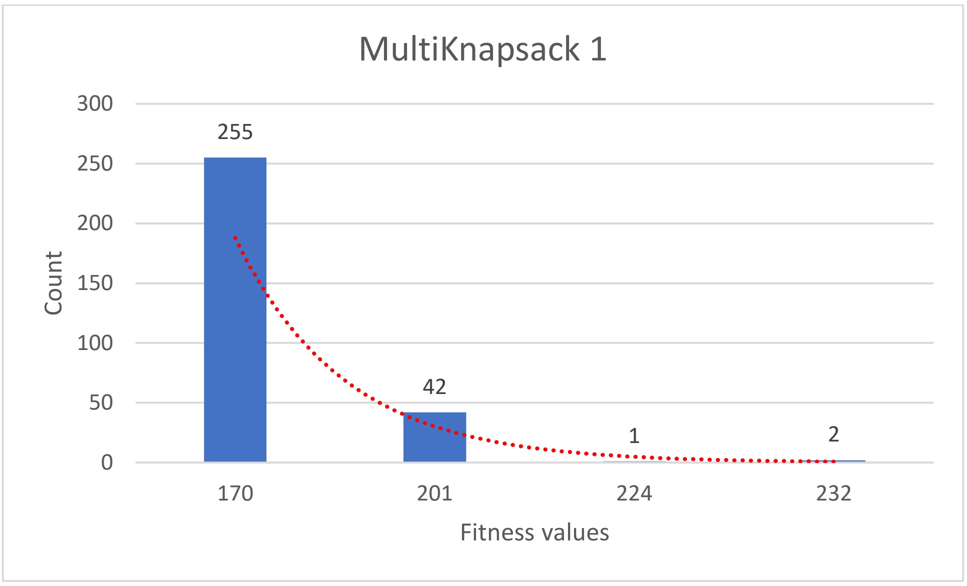
Knapsack 3

This knapsack was a very problematic one, because of the big weights of the items and the big maximum weight, that makes that the number of possible solutions were so huge than you need 5000 instances to find some solutions repeated. In this case, for example we can see 4980 different possible solutions over the 5000 instances, being the best one a chromosome with a fitness value of 14.



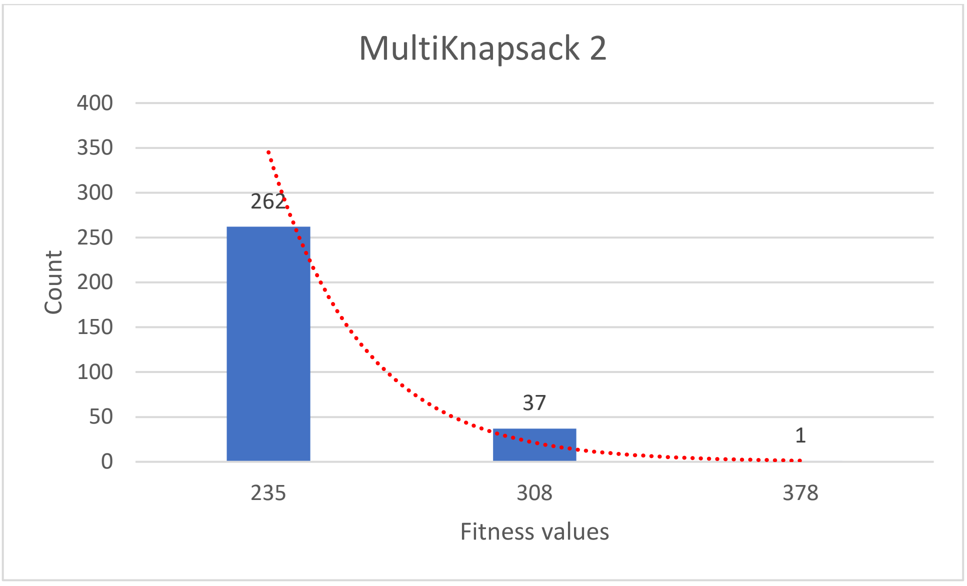
Multi-Knapsack 1

In this multi-knapsack, the best solution found has fitness 170 and were founded by 255 different chromosomes, that means that for all 300 instances there are not perfect solutions. Over the 300 instances, the genetic algorithm found 4 possible solutions.



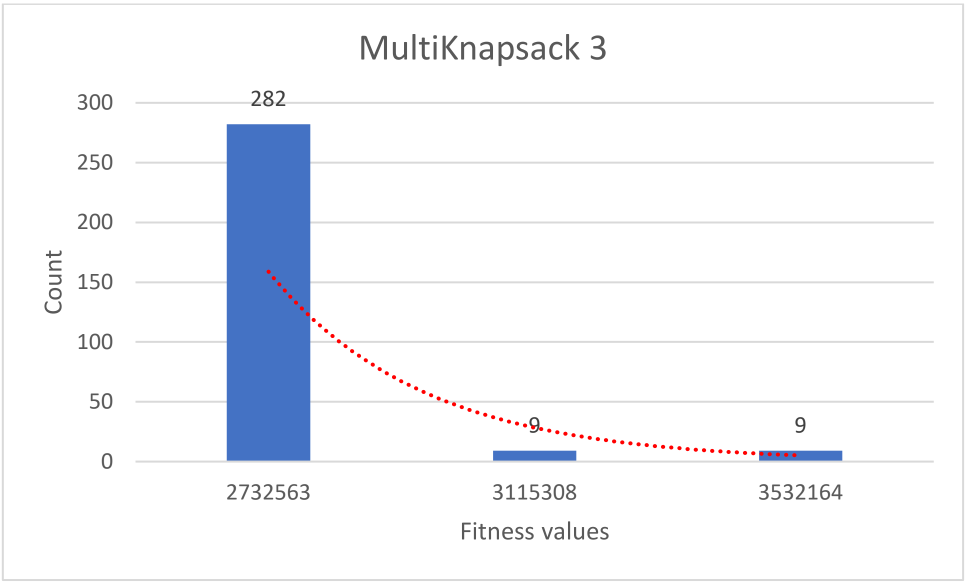
Multi-Knapsack 2

In this multi-knapsack, the best solution found has fitness 235 and were founded by 262 different chromosomes, that means that for all 300 instances there are not perfect solutions. Over the 300 instances, the genetic algorithm found 3 possible solutions.



Multi-Knapsack 3

In this multi-knapsack, the best solution found has fitness 2732563 and were founded by 282 different chromosomes, that means that for all 300 instances there are not perfect solutions. Over the 300 instances, the genetic algorithm found 3 possible solutions.



Bibliography

To help me to do this assignment, I have used:

- The slides of unit 5 of the AI course
- Code used at practice 4 that was partially solved in class
- The slides of python of the FP course
- The Wikipedia entry for the methodology of a genetic algorithm: https://en.wikipedia.org/wiki/Genetic_algorithm#Methodology
- The book called Global Optimization Algorithm -Theory and Application- by Thomas Wise, section Genetic Algorithms (page 141). You can find this book online here: http://www.it-weise.de/projects/book.pdf
  (This book is a recommended reading of the Wikipedia entry for genetic algorithms)