Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Laboratory session #04: Genetic Algorithms for combinatorial otpimization

Student: Vargas Romero Erick Efrain

Professor: Rosas Trigueros Jorge Luis

Practice completion date: November 3, 2021

Report delivery date: October 14, 2021

1

Theoretical framework

1.1 Genetic algorithms

Genetic algorithms are heuristic search and optimization techniques that simulate the process of natural selection [1].

Thus, genetic algorithms implement the optimization estrategies by simulating evolution of species through natural selection [1].

As we can see in the next figure **??**, there are some operators and parameters we need to consider.

- **Selection:** Mechanism for selecting individuals (strings) for reproduction according to their fitness (objetive function value)
- **Crossover:** Method of merging the genetic information of two individuals; if the coding is chosen properly, two good parents reproduce two good children.
- **Mutation:** In real evolution, the genetic material can be changed randomly by erroneus reproduction or other deformations of genes, e.g. by gamma radiation. In genetic algorithms, mutation can be realized as a random deformation of the strings with a certain probability. The positive effect is preservation of genetic diversity and as an effect, that local maximum can be avoided [**?**]

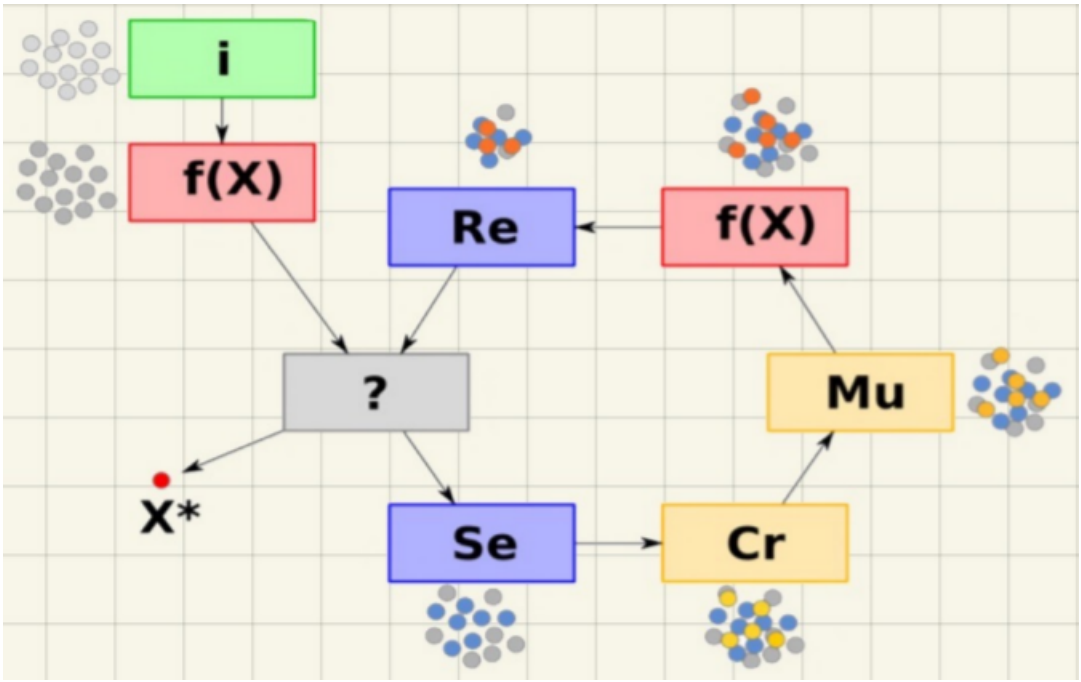

Figure 1.1: Canonical form of genetic algorithms

1.2 Functions of selection operator

- Identify the good solutions in a population.
- Make multiple copies of good solutions.
- Delete bad solutions from the population so those multiple copies of good solutions can be placed in the population.

1.3 Fitness function

Right now does not exist something like a recipe for specifying an appropiate fitness function which strongly depends on the given problem. It is, however, worth to emphasize that is necessary to provide enough information to guide the genetic algorithm to a solution. More specifically, it is not enough to define a fitness function that assigns 0 to a program which does not solve the problem and 1 to a program which solves the problem such a fitness function would correspond to a *needle-in-haystack* problem. In this sense, a profit fitness measure should be a gradual concept for judging the correctness of programs [**?**].

A fitness value is assigned to each solution depending on how close it is actually to the optimal solution of the problem [1].

1.4 Combinatorial optimization

Combinatorial optimization is the process of searching a maxima (or minima) of an objective function f whose domain is a discrete but large configuration space (as opposed an *n-dimensional* continuous space. Some simple examples of typical combinatorial optimizations are:

- The traveling salesman problem: given the (x, y) positions of n different cities, find the shortest possible path that visits each city exactly once
- Job-shop Scheduling: given a set of jobs that must be performed, and a limited set of tools with which these jobs can be performed, find a schedule for what jobs should be done when and with what tools that minimizes the total amount of time until all jobs have been completed.
- Boolean Satisfiability: assign values to a set of boolean variables in order to satisfy a given boolean expression. (A suitable objective function might be the number of satisfied clauses if the expression is a CNF formula.)

The space of possible solutions is typically too large to search exhaustively using pure brute force. In some cases, problems can be solved exactly using Branch and Bound techniques. However, in other cases no exact algorithms are feasible, and randomized search algorithms must be employed, such as:

- Random-restart hill-climbing
- Simulated annealing
- Genetic algorithms
- Tabu search

A large part of the field of Operations Research involves algorithms for solving combinatorial optimization problems [2].

1.5 Knapsack problem

Knapsack Problems are the simplest NP-hard problems in *Combinatorial Optimization*, as they maximize an objective function subject to a single resource constraint. Several variants of the classical 0–1 Knapsack Problem will be considered with respect to relaxations, bounds, reductions and other algorithmic techniques for the exact solution [3].

Suppose we are planning a hiking trip; and we are, therefore, interested in filling a knapsack with items that are considered necessary for the trip. There are n different item types that are deemed desirable; these could include bottle of water, apple, orange, sadwich, and sofort. Each item has a couple of attributes, namely a weight (or volume) and a value that quantifies the level of importance associated with each unit of that type item. Since the knapsack has a limited weight (or volume) capacity, the problem of interest is to figure out how to load the knapsack with a combination of units of the specified types of items that yields the greatest total value [4].

1.6 Traveling salesman problem

The traveling salesman problem, TSP for short, has model character in many branches of mathematics, computer science, and operations research. Heuristics, linear programming, and branch and bound, which are still the main components of today's most successful approaches to hard combinatorial optimization problems, were first formulated for the TSP and used to solve practical problem instances. When the theory of NP-completeness developed, the TSP was one of the first problems to be proven NP-hard by Karp in 1972. New algorithmic techniques have first been developed for or at least have been applied to the TSP to show their effectiveness [5].

The traveling salesman problem (also called the traveling salesperson problem or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

In the theory of computational complexity, the decision version of the TSP (where given a length L, the task is to decide whether the graph has a tour of at most L) belongs to the class of NP-complete problems. Thus, it is possible that the worst-case running time for any algorithm for the TSP increases superpolynomially (but no more than exponentially) with the number of cities.

2 Material and equipment

The necessary material for this practice is:

- A computer with the latest *Python* stable version installed
- A text editor

Or is possible to use the google site `https://colab.research.google.com/` that allows us to use a virtual machine with an *Ubuntu* operative system with *Python* installed.

3

Practice development

3.1 Genetic algorithms

To develop this practice we used the Google platform called *Colab* as this platform uses a virtual machine with linux (specifically Ubuntu) we can install some packets and of course we can verify if we have already *Python* installed. To check it we must use the command:

```
python --version
```

If we run this command in our linux terminal using *Colab* we can see the next as the result:

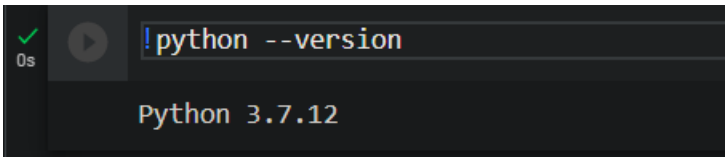

Figure 3.1: Verifying python version

3.2 Knapsack problem

For the *Knapsack problem* we must deffine some interesting things first. In the dynamic programming version we used two arrays with the values and weights of the items. For this case each chromosome represents a posible solution for this problem. For each chromosome we have only two symbols:

- 0: Represents that we do not choose the $i-th$ item.
- 1: Represents that we choose the $i-th$ item.

Once that we generate the chromosomes we must traverse that chromosome and get the *fitness value* and here we check if the current chromosome is a solution or not. Then, we choose the best chromosomes and we can proceed to make the *crossover* operation and *mutation* operation to increase the diversity of chromosomes.

The final step is to choose the chromosomes that survive in the next generation, to choose the chromosomes we make something like a "lottery", each chromosome has a specific probability to "win" but some chromosomes (the best ones) have more probability to "win". We do this because in nature is not mandatory that ever the more apt living beings are the most relevant to safeguard a species.

About code first of all as the statement of the problem says we must create randomly the items:

```
#Fill randomly v and w
def createRandomItems():
  for i in range(0, 20):
      w.append(random.random())
      v.append(random.randrange(1, 101))
```

The next interesting piece of code is the *decode_chromosome* function, because here we must process each chromosome to discover if it is a valid solution or not:

```
def decode_chromosome(chromosome):
    global L_chromosome, v, w
    Total_weight = sum([w_i * c_i for (w_i, c_i) in zip(w, chromosome)])
    Total_value = sum([v_i * c_i for (v_i, c_i) in zip(v, chromosome)])
    return (Total_value, Total_weight)
```

Of course the evaluation must be completely different to the previous practices, because here we do not have a mathematical function to evaluate instead of that we receive the weight of a chromosome and then we check if exist an excess of weight in our knapsack:

```
def f(x):
    global W
    (Total_value, Total_weight) = x
    excess = Total_weight - W
    return (Total_value if excess <= 0 else Total_value - excess * 1000)
```

Finally here we are the entire code used in this practice, as you can notice is pretty much similar to the previous practice.

```
# Knapsack problem solved with a Genetic Algorithm
w = []
v = []
W = 1

```

```
import ipywidgets as widgets


def create_button():
    button = widgets.Button(description='Next Generation',
                            disabled=False, button_style='',
                            tooltip='Next Generation', icon='check')  # 'success', 'info', 'warning', 'danger' or ''
                                                                      # (FontAwesome names without the `fa-` prefix)
    return button

def create_button_move10():
  button = widgets.Button(description = "Move 10 generations",
                          disabled = False, button_style = '',
                          tooltip = 'Move 10 generations',
                          icon = 'check')
  return button


import math
import random
import time
import matplotlib.pyplot as plt
import numpy as np
from IPython import display as display

from functools import cmp_to_key

#Fill randomly v and w
def createRandomItems():
  for i in range(0, 20):
      w.append(random.random())
      v.append(random.randrange(1, 101))

createRandomItems()

#createRandomProblem()

# Chromosomes are 4 bits long
L_chromosome = len(v)
N_chains = 2 ** L_chromosome

# Number of chromosomes
N_chromosomes = 50

# probability of mutation
prob_m = 0.5

crossover_point = int(L_chromosome / 2)

def random_chromosome():
    chromosome = []
    for i in range(0, L_chromosome):
        if random.random() < 0.1:
            chromosome.append(0)
        else:
            chromosome.append(1)
```

```python

    return chromosome


F0 = []
fitness_values = []

for i in range(0, N_chromosomes):
    F0.append(random_chromosome())
    fitness_values.append(0)

def decode_chromosome(chromosome):
    global L_chromosome, v, w

    Total_weight = sum([w_i * c_i for (w_i, c_i) in zip(w, chromosome)])
    Total_value = sum([v_i * c_i for (v_i, c_i) in zip(v, chromosome)])

    return (Total_value, Total_weight)

def f(x):
    global W
    (Total_value, Total_weight) = x
    excess = Total_weight - W
    return (Total_value if excess <= 0 else Total_value - excess * 1000)


def evaluate_chromosomes():
    global F0

    for p in range(N_chromosomes):
        v = decode_chromosome(F0[p])
        fitness_values[p] = f(v)


def compare_chromosomes(chromosome1, chromosome2):
    vc1 = decode_chromosome(chromosome1)
    vc2 = decode_chromosome(chromosome2)
    fvc1 = f(vc1)
    fvc2 = f(vc2)

    if fvc1 < fvc2:
        return 1
    elif fvc1 == fvc2:
        return 0
    else: # fvg1<fvg2
        return -1


suma = float(N_chromosomes * (N_chromosomes + 1)) / 2.

Lwheel = N_chromosomes * 10


def create_wheel():
    global F0, fitness_values

    maxv = max(fitness_values)
    acc = 0
```

```
    for p in range(N_chromosomes):
        acc += maxv - fitness_values[p]
    if acc == 0:
        return [0] * Lwheel
    fraction = []
    for p in range(N_chromosomes):
        fraction.append(float(maxv - fitness_values[p]) / acc)
        if fraction[-1] <= 1.0 / Lwheel:
            fraction[-1] = 1.0 / Lwheel

##    print fraction
    fraction[0] -= (sum(fraction) - 1.0) / 2
    fraction[1] -= (sum(fraction) - 1.0) / 2

##    print fraction
    wheel = []

    pc = 0

    for f in fraction:
        Np = int(f * Lwheel)
        for i in range(Np):
            wheel.append(pc)
        pc += 1

    return wheel


F1 = F0[:]
n = 0
def nextgeneration(b):
    global n
    display.clear_output(wait=True)
    display.display(button)
    display.display(button_move10_generations)

    F0.sort(key=cmp_to_key(compare_chromosomes))
    #print(F0)
    #print(fitness_values)
    print('Best solution so far:')
    n += 1
    print (
        n,
        F0[0],
        'f(',
        decode_chromosome(F0[0]),
        ')= ',
        f(decode_chromosome(F0[0])),
        )

    # elitism, the two best chromosomes go directly to the next generation

    F1[0] = F0[0]
    F1[1] = F0[1]
    roulette = create_wheel()

    # print (roulette)

```

```
    for i in range(0, int((N_chromosomes - 2) / 2)):

        # Two parents are selected

        p1 = random.choice(roulette)
        p2 = random.choice(roulette)

        # Two descendants are generated

        o1 = (F0[p1])[0:crossover_point]
        o1.extend((F0[p2])[crossover_point:L_chromosome])
        o2 = (F0[p2])[0:crossover_point]
        o2.extend((F0[p1])[crossover_point:L_chromosome])

        # Each descendant is mutated with probability prob_m

        if random.random() < prob_m:
            o1[int(round(random.random() * (L_chromosome - 1)))] ^= 1
        if random.random() < prob_m:
            o2[int(round(random.random() * (L_chromosome - 1)))] ^= 1

        # The descendants are added to F1

        F1[2 + 2 * i] = o1
        F1[3 + 2 * i] = o2

    # The generation replaces the old one

    F0[:] = F1[:]
    evaluate_chromosomes()

def move10_generations(param):
  for i in range(0, 10):
    nextgeneration(param)

xmax = 400
ymax = 400

button = create_button()
button.on_click(nextgeneration)
button_move10_generations = create_button_move10()
button_move10_generations.on_click(move10_generations)

display.display(button)
display.display(button_move10_generations)

F0.sort(key=cmp_to_key(compare_chromosomes))
evaluate_chromosomes()
```

3.3 Traveling Salesman Problem

The other exercise is solve the *Traveling Salesman Problem* using a genetic algorithm. First of all as we did in the previous exercise we must define what is a chromosome?. In this case we define a chromosome as a sequence of integers where each integer is the representation of a vertex between nodes. The simplest case to understand this is:

$$[x_1, x_2, \ldots, x_{n-1}, x_n]$$

The previous array means, the city or node x_1 is connected with the node x_2 and the node x_2 is connected with the node x_n and so on.

Other part of this problem that we must define is the *fitness function*, in our case we must change it because our target is to minimize the total distance in our graph. Other interesting operation is the crossover, here we have a lot of different possible algorithms but we choose the *Partially-mapped crossover* or *PMX*. Finally again we have the mutation operations and the process of selection to define if a chromosome survives in the next generation is the same as in the previous problem.

In this problem we used seven cities from: `https://www.infoplease.com/world/travel-transportation/air-distances-between-world-cities-statute-miles`, more specifically we choose the cities: Mexico city, Hong Kong, Honolulu, Lisbon, London, Los Angeles and Manali. If we build our adjacency matrix we got tbl. 3.1 as the result.

	Mexico city	Hong Kong	Honolulu	Lisbon	London	Los Angeles	Manali
Mexico city	0	8,782	3,779	5,390	5,550	1,589	8,835
Hong Kong	8,782	0	5,549	6,853	5,982	7,195	693
Honolulu	3,779	5,549	0	7,820	7,228	2,574	5,299
Lisbon	5,390	6,853	7,820	0	985	5,621	7,546
London	5,550	5,982	7,228	985	0	5,382	6,672
Los Angeles	1,589	7,195	2,574	5,621	5,382	0	7,261
Manali	8,835	693	5,299	7,546	6,672	7,261	0

Table 3.1: Representation of our adjacency matrix

In each city you can move to the other ones and it makes it a complex problem, if we represent the previous table as a graph we get the img. 3.2 as the result.

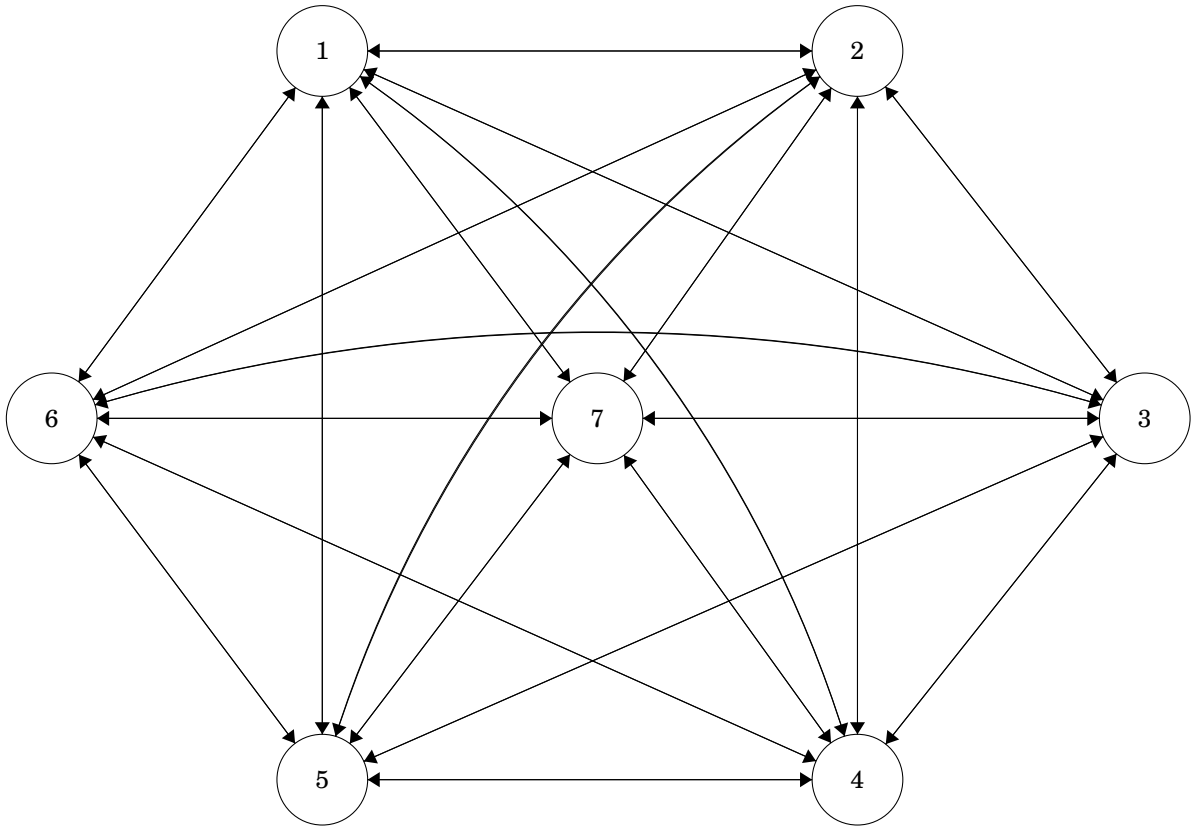

Figure 3.2: General graph representation of the problem

Fortunatelly we can use again the code of the *Knapsack problem* but of course we must to adapt the code to solve the traveling salesman problem.

First, we must represent our graph, as we mentioned before we will use an adjacency matrix where en each element of the matrix we store the distance between the $i-th$ city and the $j-th$ city.

```
#Remember we are solving the traveling salesman problem so we need to use distances
    between cities
#Create the adjacency matrix
cities = [[0,8782,3779,5390,5550,1589,8835],
          [8782,0,5549,6853,5982,7195,693],
          [3779,5549,0,7820,7228,2574,5299],
          [5390,6853,7820,0,985,5621,7546],
          [5550,5982,7228,985,0,5382,6672],
          [1589,7195,2574,5621,5382,0,7261],
          [8835,693,5299,7546,6672,7261,0]]
```

To make easier this problem we generated all the posible permutations of an initial chromosome of seven elements this function takes $O(n!)$ time and memory.

```
def generateAllPermutations(numberOfNodes):
```

```
  #Each city is a node of a graph the total number of permutations is given by n!
    where n is the number of nodes
  numberOfPermutations = math.factorial(numberOfNodes)
  #We get all the permutations
  for i in range(0, numberOfPermutations):
    chromosome = nextPermutation(listOfPossibleChromosomes[i])
    listOfPossibleChromosomes.append(chromosome)
```

The next step is, change our function called $f(x)$; this function calculates the distance. First of all we must be sure that we have a valid vertex in other words we can not go from the city a to the city a it means that we do not move from the same city but if we do not have any problem with that we accumulate the distance from the $i-th$ city to the $i-th+1$ city if the condition was false we need discard the current candidate to be a solution.

```
#We get the total distance
def f(x):
  distancePath = 0
  for i in range (0, len(x) - 1):
    distance = cities[x[i]][x[i + 1]]
    if distance != 0:
      distancePath += distance
    else:
      distancePath = math.inf
      break
  return distancePath
```

Finally the most interesting function of this problem, the crossover. As we mentioned before we implemented the *Partially-mapped crossover* or *PMX*, to solve it as easier as possible we first build a candidate to be a child of the ancestors called *ancestor1* and *ancestor2*, in this step is highly posible to have an invalid solution, it means a repeated city in our chromosome.

```
#Function that makes the crossover process using the PMX algorithm
def crossover(ancestor1, ancestor2):
  #We are "spliting" our chromosomes in two parts of random size in order to "mix"
    each part of the information
  crossPoint1 = random.randint(0, len(ancestor1) - 2)
  crossPoint2 = random.randint(crossPoint1 + 1, len(ancestor1) - 1)
  #Now we make a copy of the "left piece" of the chromosomes received
  middleCrossA1 = ancestor1[crossPoint1:crossPoint2]
  middleCrossA2 = ancestor2[crossPoint1:crossPoint2]
  middleCrossSize = crossPoint2 - crossPoint1
  #Creating partial new chromosomes creating a relationship between the pieces that
    we got from crossPoint1 and crossPoint2
  #And mixing them with the other ancestor graphicaly something like (x x x | a b c
    | x x)
  partialChild1 = ancestor1[:crossPoint1] + middleCrossA2 + ancestor1[crossPoint2:]
  partialChild2 = ancestor2[:crossPoint1] + middleCrossA1 + ancestor2[crossPoint2:]
```

But now we have a problem, we must avoid to have repeated elements in the chromosome, so we can use the mapping technique to get which elements can not use and of course we can map the relationship between elements.

```
#We can use a data structure to know as fast as possible if the element that we
    choose is already in use (A set)
  valuesAlreadyExist1 = set()
  valuesAlreadyExist2 = set()
  relations = {}
  for index in range(0, middleCrossSize):
```

```
    u = middleCrossA1[index]
    v = middleCrossA2[index]
    valuesAlreadyExist1.add(v)
    valuesAlreadyExist2.add(u)
    #We add the relations between elements to find as fast as possible wich element
    we can use
    if not u in relations:
      relations[u] = [v]
    else:
      relations[u].append(v)

    if not v in relations:
      relations[v] = [u]
    else:
      relations[v].append(u)
```

Then, we can try to get a valid child but to solve this we can create other function inside the crossover function that allow us to evaluate if we need to try to search an available element

```
def getValidChild(partialChild, valuesAlreadyExist):
    partialChildSize = len(partialChild)
    #Checking if the current chromosome contains unique and valid values
    for index in range(0, partialChildSize):
      if (index < crossPoint1 or index >= crossPoint2):
        currElement = partialChild[index]
```

Now we have another problem, how can we get an available element?, we can create another function that help us with that problem. The *findValidValue* function uses a very similar idea of a BFS. We get the children of the current node and we add it in the queue but if the current element is not already visited it means it is an available item, so we take it as the answer and we finish our exhaustive search.

```
#For this function we'll use a similar idea to a BFS to try to find the valid
    element that we can use
#But we are asuming that for ever we will find a valid solution or in other words
#We're asuming that the relations have an "end"
def findValidValue(key):
  if key in valuesAlreadyExist:
    queue = []
    queue.append(key)
    while len(queue) != 0:
      u = queue.pop(0)
      for v in relations[u]:
        if not v in valuesAlreadyExist:
          valuesAlreadyExist.add(v)
          return v
        queue.append(v)
  return key
```

Then, we can find a valid value to the $i-th$ element of our partial valid chromosome and finally return it.

```
def getValidChild(partialChild, valuesAlreadyExist):
    #Rest of the code ...

        validValue = findValidValue(currElement)
        partialChild[index] = validValue
    return partialChild
```

Finally, we get the new both children and we return them.

```
#Function that makes the crossover process using the PMX algorithm
def crossover(ancestor1, ancestor2):
  #Rest of the code ...

  validChild1 = getValidChild(partialChild1, valuesAlreadyExist1)
  validChild2 = getValidChild(partialChild2, valuesAlreadyExist2)

  return validChild1, validChild2
```

Here you can see the entire code of this problem, we are sure it can help you to understand easily the idea behind this solution:

```
#Traveling salesman problem with a GA

import math
import random
import numpy as np
from IPython import display as display
from functools import cmp_to_key
import ipywidgets as widgets

#Number of chromosomes
nChromosomes = 100

#Probablity of mutation
probM = 0.5

#Chromosomes list
F0 = []

#Chromosomes fitness value
fitnessValues = []

#Remember we are solving the traveling salesman problem so we need to use distances
    between cities
#Create the adjacency matrix
cities = [[0,8782,3779,5390,5550,1589,8835],
          [8782,0,5549,6853,5982,7195,693],
          [3779,5549,0,7820,7228,2574,5299],
          [5390,6853,7820,0,985,5621,7546],
          [5550,5982,7228,985,0,5382,6672],
          [1589,7195,2574,5621,5382,0,7261],
          [8835,693,5299,7546,6672,7261,0]]

def createButton():
  button = widgets.Button(
      description = "Next generation",
      disabled = False,
      button_style = '',
      tooltip = 'Next generation',
      icon = 'check'
  )
  return button

def createButton10Generations():
  button = widgets.Button(
      description = "Move 10 generations",
```

```
      disabled = False,
      button_style = '',
      tooltip = 'Move 10 generations',
      icon = 'check'
  )
  return button

def nextPermutation(arr):
  arrCopy = arr.copy()
  wasFound = False
  #At least we need two numbers to get a permutation
  i = len(arrCopy) - 2
  while i >= 0:
    if arrCopy[i] < arrCopy[i + 1]:
      wasFound = True
      break
    i -= 1

  if not wasFound:
    arrCopy.sort()
  else:
    pivotIndex = findPivot(arrCopy, arrCopy[i])
    arrCopy[i], arrCopy[pivotIndex] = arrCopy[pivotIndex], arrCopy[i]
    arrCopy[i + 1:] = arrCopy[i + 1:][:: -1]
  return arrCopy

def findPivot(arr, element):
  ans = -1
  index = 0
  for i in range(index, len(arr)):
    if arr[i] > element:
      if ans == -1:
        ans = element
        index = i
      else:
        ans = min(ans, arr[i])
        index = i
  return index

listOfPossibleChromosomes = [[0, 1, 2, 3, 4, 5, 6]]

def generateAllPermutations(numberOfNodes):
  #Each city is a node of a graph the total number of permutations is given by n!
    where n is the number of nodes
  numberOfPermutations = math.factorial(numberOfNodes)
  #We get all the permutations
  for i in range(0, numberOfPermutations):
    chromosome = nextPermutation(listOfPossibleChromosomes[i])
    listOfPossibleChromosomes.append(chromosome)

    #print(listOfPossibleChromosomes)

def getAllChromosomes(numberOfNodes):
  upperBound = math.factorial(numberOfNodes)
  for i in range(0, nChromosomes):
    F0.append(listOfPossibleChromosomes[random.randrange(0, upperBound)])
    fitnessValues.append(0)

```

```
generateAllPermutations(7)
getAllChromosomes(7)

#We get the total distance
def f(x):
  distancePath = 0
  for i in range (0, len(x) - 1):
    distance = cities[x[i]][x[i + 1]]
    if distance != 0:
      distancePath += distance
    else:
      distancePath = math.inf
      break
  return distancePath

#Or in other words our fitness function
def evaluateChromosomes():
  global F0
  for p in range(nChromosomes):
    fitnessValues[p] = f(F0[p])

def compareChromosomes(chromosome1, chromosome2):
  fResultChromosome1 = f(chromosome1)
  fResultChromosome2 = f(chromosome2)

  if fResultChromosome1 > fResultChromosome2:
    return 1
  elif fResultChromosome1 == fResultChromosome2:
    return 0
  #fResultChromosome1 < fResultChromosome2 otherwise
  return -1

#Function that makes the crossover process using the PMX algorithm
def crossover(ancestor1, ancestor2):
  #We are "spliting" our chromosomes in two parts of random size in order to "mix" each part of the information
  crossPoint1 = random.randint(0, len(ancestor1) - 2)
  crossPoint2 = random.randint(crossPoint1 + 1, len(ancestor1) - 1)
  #Now we make a copy of the "left piece" of the chromosomes received
  middleCrossA1 = ancestor1[crossPoint1:crossPoint2]
  middleCrossA2 = ancestor2[crossPoint1:crossPoint2]
  middleCrossSize = crossPoint2 - crossPoint1
  #Creating partial new chromosomes creating a relationship between the pieces that we got from crossPoint1 and crossPoint2
  #And mixing them with the other ancestor graphicaly something like (x x x | a b c | x x)
  partialChild1 = ancestor1[:crossPoint1] + middleCrossA2 + ancestor1[crossPoint2:]
  partialChild2 = ancestor2[:crossPoint1] + middleCrossA1 + ancestor2[crossPoint2:]
  #We can use a data structure to know as fast as possible if the element that we choose is already in use (A set)
  valuesAlreadyExist1 = set()
  valuesAlreadyExist2 = set()
  relations = {}
  for index in range(0, middleCrossSize):
    u = middleCrossA1[index]
    v = middleCrossA2[index]
    valuesAlreadyExist1.add(v)
    valuesAlreadyExist2.add(u)
```

```
    #We add the relations between elements to find as fast as possible wich element we can use
    if not u in relations:
      relations[u] = [v]
    else:
      relations[u].append(v)

    if not v in relations:
      relations[v] = [u]
    else:
      relations[v].append(u)

  def getValidChild(partialChild, valuesAlreadyExist):
    partialChildSize = len(partialChild)
    #Checking if the current chromosome contains unique and valid values
    for index in range(0, partialChildSize):
      if (index < crossPoint1 or index >= crossPoint2):
        currElement = partialChild[index]

        #For this function we'll use a similar idea to a BFS to try to find the valid element that we can use
        #But we are asuming that for ever we will find a valid solution or in other words
        #We're asuming that the relations have an "end"
        def findValidValue(key):
          if key in valuesAlreadyExist:
            queue = []
            queue.append(key)

            while len(queue) != 0:
              u = queue.pop(0)
              for v in relations[u]:
                if not v in valuesAlreadyExist:
                  valuesAlreadyExist.add(v)
                  return v
                queue.append(v)
          return key

        validValue = findValidValue(currElement)
        partialChild[index] = validValue
    return partialChild

  validChild1 = getValidChild(partialChild1, valuesAlreadyExist1)
  validChild2 = getValidChild(partialChild2, valuesAlreadyExist2)

  return validChild1, validChild2

#It is a sum using the gauss sum
suma = float(nChromosomes * (nChromosomes + 1)) / 2
lWheel = nChromosomes * 10

def createWheel():
  global F0, fitnessValues
  maxValue = max(fitnessValues)
  accumulator = 0
  for p in range(nChromosomes):
    accumulator += maxValue - fitnessValues[p]
  if accumulator == 0:
```

```
    return [0]*lWheel
  fraction = []
  for p in range(nChromosomes):
    fraction.append(float(maxValue - fitnessValues[p]) / accumulator)
    if fraction[-1] <= 1.0 / lWheel:
      fraction[-1] = 1.0 / lWheel
  fraction[0] -= (math.fsum(fraction)-1.0) / 2
  fraction[1] -= (math.fsum(fraction)-1.0) / 2

  wheel = []
  pc = 0
  for f in fraction:
    Np = int(f * lWheel)
    for i in range(Np):
      wheel.append(pc)
    pc += 1
  return wheel

F1 = F0[:]
n = 0

#Generate the next generation of chromosomes
def nextGeneration(b):
    global n
    display.clear_output(wait = True)
    display.display(button)
    display.display(button10Generations)
    F0.sort(key = cmp_to_key(compareChromosomes) )
    print( "Best solution so far:")
    n += 1
    print( n,F0[0],"\nf(",F0[0],") = ", f(F0[0]) )

    F1[0]=F0[0]
    F1[1]=F0[1]

    roulette = createWheel()

    for i in range(0, int((nChromosomes- 2 ) / 2)):

        ancestor1 = random.choice(roulette)
        ancestor2 = random.choice(roulette)

        offsprings = crossover(F0[ancestor1], F0[ancestor2])
        offspring1 = offsprings[0]
        offspring2 = offsprings[1]

        if random.random() < probM:
            pos = random.randrange(0,7)
            pos2 =  random.randrange(0,7)
            offspring1[pos], offspring1[pos2] = offspring1[pos2], offspring1[pos]
        if random.random() < probM:
            pos = random.randrange(0,7)
            pos2 =  random.randrange(0,7)
            offspring2[pos], offspring2[pos2] = offspring2[pos2], offspring2[pos]
        F1[2 + 2 * i] = offspring1
        F1[3 + 2 * i] = offspring2

    F0[:] = F1[:]
```

```
    evaluateChromosomes()


def move10Generations(param):
  for i in range(0, 10):
    nextGeneration(param)

button = createButton()
button.on_click(nextGeneration)
display.display(button)

button10Generations = createButton10Generations()
button10Generations.on_click(move10Generations)
display.display(button10Generations)


F0.sort( key=cmp_to_key(compareChromosomes) )
evaluateChromosomes()
```

4

Screens, graphs and diagrams

4.1 Knapsack problem

As you know for the *Knapsack problem* we generate random weights and values if we run the program you will see something different each time. You can see an example in the img. 4.1, img. 4.2, img. 4.3, img. 4.4, img. 4.5 the algorithm solution continues changing and looks like the current solution is the best but if we iterate until generation 200 img. 4.6 as you can see the value changes again, but if we continue until generation 300 img. 4.7 stills being the same value, so probably the solution with value 304 is the best one so far.

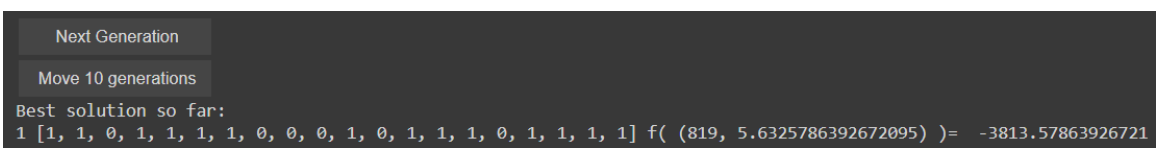

Figure 4.1: KSP simulation generation 1

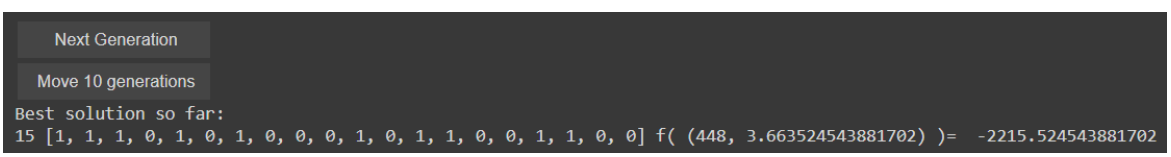

Figure 4.2: KSP simulation generation 15

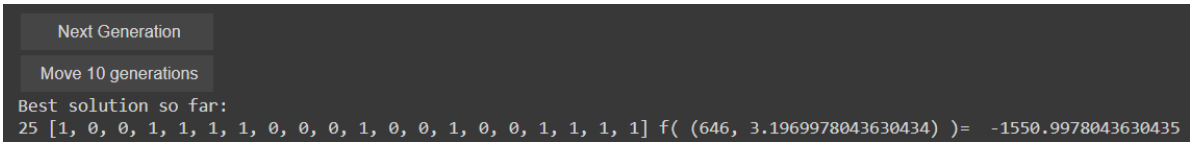

Figure 4.3: KSP simulation generation 25

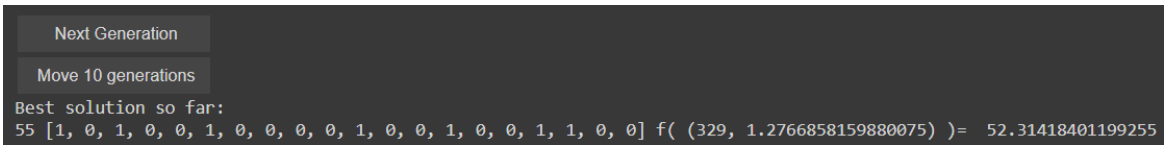

Figure 4.4: KSP simulation generation 55

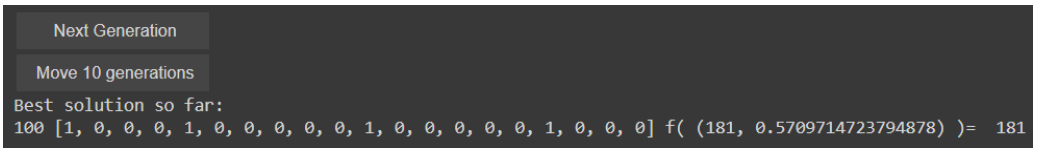

Figure 4.5: KSP simulation generation 100

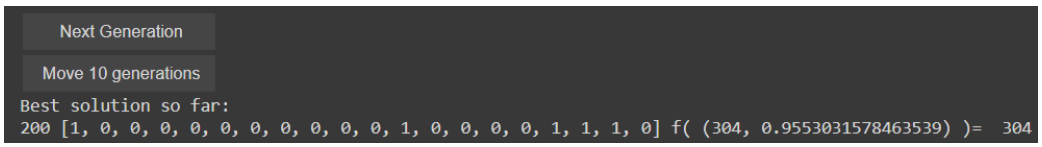

Figure 4.6: KSP simulation generation 200

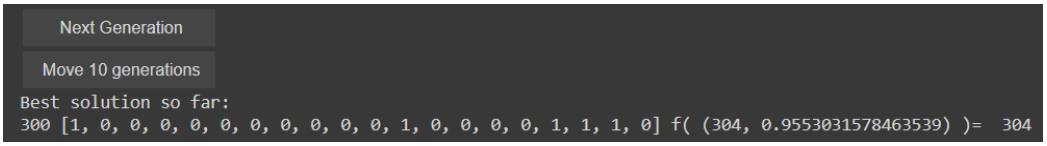

Figure 4.7: KSP simulation generation 300

4.2 Traveling salesman problem

For the *Traveling salesman problem* we have a similar situation than in the *Knapsack problem* we are generating randomly chromosomes but the data stills being the same so we must get the same solution ever. As you can see in this problem we got the solution too quickly, first in the img. 4.8 we have the first generation and in the generation five img. 4.9, we got the value 16,530 and if we continue iterating until the generation 100 the value stills being the same img. 4.10.

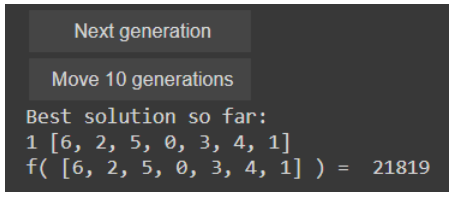

Figure 4.8: TSP simulation generation 1

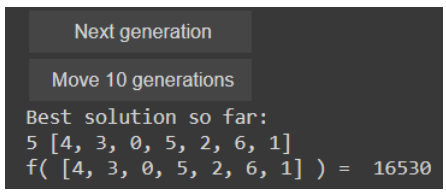

Figure 4.9: TSP simulation generation 5

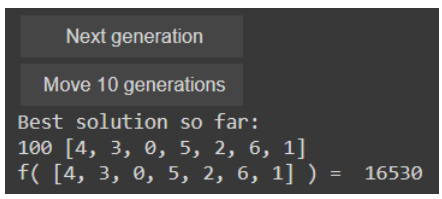

Figure 4.10: TSP simulation generation 100

5 Conclusions

This practice has been too interesting because first we merged the *knapsack problem* first understanding how it works using *dynamic programming* and then solving the same problem but now with *Genetic algorithms*. The *Traveling salesman problem* was the most complicated problem in this practice, first of all because is mandatory to understand as much as possible the algorithm that we will in the crossover, at lest for me was complicated because try to find the best way to tackle the problem was hard, first trying to solve it by brute force using lists, and then optimizing a little bit using the mapping technique and some concepts of graphs like BFS.

Bibliography

[1] R. K. Bhattacharjya *Introduction to Genetic Algorithms*. Department of Civil Engineering. November, 2013. Accessed on October 25th, 2021. Available online: https://www.iitg.ac.in/rkbc/CE602/%20CE602/Genetic%20Algorithms.pdf

[2] Scott. *What is Combinatorial Optimization?*. Accessed on October 30th, 2021. Available online:https://www.cs.cmu.edu/afs/cs.cmu.edu/project/learn-43/lib/photoz/.g/web/glossary/comb.html

[3] David P. Paolo T. *Handbook of combinatorial optimizations*. Accessed on October 30th, 2021. Available online:https://link.springer.com/chapter/10.1007/978-1-4613-0303-9_5

[4] F.S. Hillier and G. J. Lieberman. *Introduction to operations Research*. 6th edition. McGraw-Hill, 1995. Accessed on October 30th, 2021. Available online: https://personal.utdallas.edu/~scniu/OPRE-6201/documents/DP3-Knapsack.pdf

[5] M. Jünger, G. Reinelt and G. Rinaldi *The traveling salesman problem*. Accessed on October 30th, 2021. Available online: https://www.sciencedirect.com/science/article/abs/pii/S0927050705801215