Instituto Politécnico Nacional
Escuela Superior de Cómputo
Evolutionary Computing
4 GA for Combinatorial Optimization
Martínez Coronel Brayan Yosafat
Rosas Trigueros Jorge Luis
24/10/2021
8/10/2021

## Theorical framework

In this practice, the use of Genetic Algorithms is made over two combinatorial problems: Knapsack Problem and Travelling Salesman Problem, one has been discussed in earlier practices, but it was solved using dynamic programming.

First, defining the problems is the most important part, according to Wikipedia, Knapsack problem is defined as: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible [1].

On the other hand, TSP, or travelling salesman problem questions us: 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? [2]

Genetic algorithms are inspired by natural selection, they belong to the evolutionary algorithms class. Used for optimization with good solutions and search problems, they relied on biological inspired operators such as mutation, crossover, and selection. Based on a population of individuals representing candidates for solutions, created by random methods, and then iterated over time, we finally come to a solution (or more) to resolve the problem. Best candidates are

selected every iteration, but these rules are general and can be modified according to the problem to resolve [3].

In general terms, with a population, at the end of each iteration (called generation), individuals are evaluated with a function (called fitness function, because it checks how much an individual fits the solution) and then, using biological inspired operators, the next generation is created. Over the time, result come to improve and converge (if there is a solution) [4].

In this practice, for instance, we modify the mutation and crossover for the TSP, in the development we discuss about it in more detail.

## Material and equipment

In this practice, DataSpell with Jupiter Notebooks, Conda and Python were used for development, all of them were run in a personal computer which has an Intel Core i7 with Windows 10.

## Practice development

### Knapsack problem

Just as in the practice number 3, we define in block of code for the sake of comprehension:

```
import numpy as np

MOCHILA = 1
K_PENALIZACION = 20

def evaluar(peso: np.ndarray, valor: np.ndarray):
    exceso = MOCHILA - peso
    evaluacion = valor.copy()
    evaluacion[exceso < 0] = evaluacion[exceso < 0] -
np.abs(evaluacion[exceso < 0] * exceso[exceso < 0]) - K_PENALIZACION
    return evaluacion
```

Code 1. Fitness function definition

```
NUM_OBJETOS = 20
VALORES = np.random.randint(low=1, high=101, size=NUM_OBJETOS)
```

```
PESOS = np.random.random(size=NUM_OBJETOS)      # Puede regresar 0
PESOS[PESOS == 0] = 0.1

class Cromosoma:
    """
    Se conforma por una lista que sirve como índice lógico para PESOS
y VALORES
    """
    def __init__(self, objetos = None):
        if objetos is None:
            self.list = np.random.choice([0, 1], size=NUM_OBJETOS)
        else:
            self.list = objetos

    def __str__(self):
        return str(self.list)


    def get_peso(self):
        return np.sum(PESOS[self.list == 1])

    def get_valor(self):
        return np.sum(VALORES[self.list == 1])

    @staticmethod
    def crossover(c1, c2):
        mitad = int(NUM_OBJETOS / 2)
        l1 = np.append(c1.list[0:mitad], c2.list[mitad:], axis=None)
        l2 = np.append(c2.list[0:mitad], c1.list[mitad:], axis=None)
        return [Cromosoma(l1), Cromosoma(l2)]

    @staticmethod
    def mutar(c):
        index = np.random.randint(low=0, high=NUM_OBJETOS)
        c.list[index] = 1 - c.list[index]
```

Code 2. Chromosome class definition

```
K_POBLACION = 8
K_BASE = 2
K_PROBABILIDAD_MUTACION = 0.5

def presion_selectiva(poblacion: list[Cromosoma]) -> list[Cromosoma]:
    # Evaluación y búsqueda del mejor
    c_pesos = np.array([c.get_peso() for c in poblacion])
    c_valores = np.array([c.get_valor() for c in poblacion])
    evaluacion = evaluar(c_pesos, c_valores)

    best = evaluacion.argmax()
    print("Best so far:")
    print(f"Combination: {poblacion[best]}")
    print(f"Weight: {c_pesos[best]}")
    print(f"Value: {c_valores[best]}")
    print(f"Evaluation: {evaluacion[best]}")
```

```
    # Cálculo de probabilidades
    indice_ordenado = (-evaluacion).argsort()
    ruleta = []
    potencia = K_POBLACION

    for i in indice_ordenado:
        probabilidad = K_BASE ** potencia
        ruleta.extend([i] * probabilidad)
        potencia -= 1

    # Nueva generación
    nueva = list[Cromosoma]()
    nueva.append(poblacion[indice_ordenado[0]])
    nueva.append(poblacion[indice_ordenado[1]])

    for i in range(1, int(K_POBLACION/2)):
        c1 = poblacion[np.random.choice(ruleta)]
        c2 = poblacion[np.random.choice(ruleta)]
        hijos = Cromosoma.crossover(c1, c2)

        for hijo in hijos:
            if np.random.choice([True, False],
p=[K_PROBABILIDAD_MUTACION, 1-K_PROBABILIDAD_MUTACION]):
                Cromosoma.mutar(hijo)

        nueva.extend(hijos)

    return nueva
```

Code 3. Selection pressure definition

And just, with the simplicity of OOP design, the run code is the same:

```
poblacion = list[Cromosoma]()
nueva_poblacion = list[Cromosoma]()
generacion = 0
```

```
if len(nueva_poblacion) == 0:
    poblacion = [Cromosoma() for _ in range(0, K_POBLACION)]
else:
    poblacion = nueva_poblacion

print('Generation', generacion)
nueva_poblacion = presion_selectiva(poblacion)
generacion += 1
```

Code 4 and 5. Resetting and running codes

As we observe, it is pretty similar from problems of optimization, the fitness function is one the greatest changes comparing to Ackley function and Rastrigin

function. That is the beauty of this kind of design, it is independent of its parts, and, faster to develop over the time. Of course, the first time takes a considerable amount of time, but in these two problems the development time was much faster.

```
Generation 0
Best so far:
Combination: [0 0 1 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0]
Weight: 1.9826077913139266
Value: 291
Evaluation: -14
```

```
Generation 1
Best so far:
Combination: [0 0 1 0 1 1 0 0 0 0 0 0 0 0 1 0 0 0 0]
Weight: 1.8794489302184947
Value: 234
Evaluation: 8
```

```
Generation 2
Best so far:
Combination: [0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0]
Weight: 1.1637854810608363
Value: 276
Evaluation: 210
```

```
Generation 3
Best so far:
Combination: [0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0]
Weight: 0.8901989212245465
Value: 223
Evaluation: 223
```

Fig. 1, 2, 3 and 4. Generations 0, 1, 2 and 3

We don't remove chromosomes who have a weight greater than 1, but the greater, the bigger the penalization, this is good for individuals who are near of the answer.

```
Generation 9
Best so far:
Combination: [0 1 0 0 0 1 0 0 0 0 0 0 1 0 1 1 0 0 0 0]
Weight: 0.976136194500251
Value: 314
Evaluation: 314
```

Fig. 5 Results of the Knapsack problem

In the next generations it keeps the same way, some individuals try to find other solutions but hey return at the next generation.

## Travelling Salesman Problem

Recalling about the definition of crossover, we need to do this in order to satisfy a condition about the problem, first, the chromosome in this time, is not binary, this is really important because we have been working just with binary chromosomes, now, the chromosome is a list of integer numbers which represents each one, a city. The cities taken in consideration for this exercise are:

| | Mexico | Montreal | Moscow | N. York | Paris | Rio | Rome |
|---|---|---|---|---|---|---|---|
| Mexico | 0 | 2318 | 6663 | 2094 | 5716 | 4771 | 6366 |
| Montreal | 2318 | 0 | 4386 | 320 | 3422 | 5097 | 4080 |
| Moscow | 6663 | 4386 | 0 | 4065 | 1544 | 7175 | 1474 |
| N. York | 2094 | 320 | 4065 | 0 | 3624 | 4817 | 4281 |
| Paris | 5716 | 3422 | 1544 | 3624 | 0 | 5699 | 697 |
| Rio | 4771 | 5097 | 7175 | 4817 | 5699 | 0 | 5684 |
| Rome | 6366 | 4080 | 1474 | 4281 | 697 | 5684 | 0 |

Table 1. Distances between some cities [5]

Now, crossover cannot repeat numbers in the list, so, we use a partial-mapped crossover we take three adjacent cities, and the rest is (or at least we try) crossover. In the code is clearer [6].

Something similar happens to mutation, now we have exchange mutation and displacement mutation, the first one just takes two indexes and change each other position, whereas the second take three and move them to another position [6].

```
import numpy as np

def fitness(distancias: np.ndarray):
    return np.sum(distancias, axis=1)
```

Code 6. Fitness function definition

```
NUM_CIUDADES = 7
BASE = np.array(range(0, NUM_CIUDADES))
RNG = np.random.default_rng()
DISTANCIAS = np.array([
    [0, 2318, 6663, 2094, 5716, 4771, 6366],
    [2318, 0, 4386, 320, 3422, 5097, 4080],
    [6663, 4386, 0, 4065, 1544, 7175, 1474],
    [2094, 320, 4065, 0, 3624, 4817, 4281],
    [5716, 3422, 1544, 3624, 0, 5699, 697],
    [4771, 5097, 7175, 4817, 5699, 0, 5684],
    [6366, 4080, 1474, 4281, 697, 5684, 0]
])

class Cromosoma:
    """
    Se conforma por una lista que representa el orden en el que
recorre las ciudades, cada número no debe repetirse
    """
    def __init__(self, ciudades=None):
        if ciudades is None:
            self.list = RNG.permutation(BASE)
        else:
            self.list = ciudades


    def __str__(self):
        return str(self.list)


    def get_distancias(self):
        distancias = [DISTANCIAS[self.list[i]][self.list[i+1]] for i
in range(0, NUM_CIUDADES-1)]
        distancias.append(DISTANCIAS[self.list[-1]][self.list[0]])  #
Para regresar al origen
        return np.array(distancias)


    @staticmethod
    def crossover_parcial(c1, c2):
        offspring_1 = -np.ones_like(c1.list)
```

```
        offspring_2 = -np.ones_like(c2.list)
        index = np.random.randint(0, high=NUM_CIUDADES-3)

        equivalencia_1 = c2.list[index:index+3].copy()
        equivalencia_2 = c1.list[index:index+3].copy()
        offspring_1[index:index+3] = equivalencia_1
        offspring_2[index:index+3] = equivalencia_2

        for i in range(0, NUM_CIUDADES):
            iterador = c1.list[i]

            if offspring_1[i] != -1:
                continue
            elif iterador in offspring_1:
                while iterador in offspring_1:
                    iterador = int(equivalencia_2[equivalencia_1 ==
iterador])

            offspring_1[i] = iterador

        for i in range(0, NUM_CIUDADES):
            iterador = c2.list[i]

            if offspring_2[i] != -1:
                continue
            elif iterador in offspring_2:
                while iterador in offspring_2:
                    iterador = int(equivalencia_1[equivalencia_2 ==
iterador])

            offspring_2[i] = iterador

        return [Cromosoma(offspring_1), Cromosoma(offspring_2)]

    @staticmethod
    def mutacion_desplazada(c):
        desde = np.random.randint(0, high=NUM_CIUDADES-3)
        hasta = np.random.randint(0, high=NUM_CIUDADES-3)
        temporal = c.list[hasta:hasta+3].copy()
        c.list[hasta:hasta+3] = c.list[desde:desde+3]
        c.list[desde:desde+3] = temporal

    @staticmethod
    def mutacion_intercambio(c):
        i = np.random.randint(low=0, high=NUM_CIUDADES)
        j = np.random.randint(low=0, high=NUM_CIUDADES)
        temporal = c.list[i]
        c.list[i] = c.list[j]
        c.list[j] = temporal
```

Code 7. Chromosome class definition

Observe that we change a lot the methods, the reason is the list contains integers that must be uniques.

```
K_POBLACION = 8
K_BASE = 2
K_PROBABILIDAD_MUTACION = 0.5

def presion_selectiva(poblacion: list[Cromosoma]) -> list[Cromosoma]:
    # Evaluación y búsqueda del mejor
    distancias = np.array([c.get_distancias() for c in poblacion])
    evaluacion = fitness(distancias)

    best = evaluacion.argmin()
    print("Best so far:")
    print(f"Combination: {poblacion[best]}")
    print(f"Distances: {distancias[best]}")
    print(f"Evaluation: {evaluacion[best]}")

    # Cálculo de probabilidades
    indice_ordenado = evaluacion.argsort()
    ruleta = []
    potencia = K_POBLACION

    for i in indice_ordenado:
        probabilidad = K_BASE ** potencia
        ruleta.extend([i] * probabilidad)
        potencia -= 1

    # Nueva generación
    nueva = list[Cromosoma]()
    nueva.append(poblacion[indice_ordenado[0]])
    nueva.append(poblacion[indice_ordenado[1]])

    for i in range(1, int(K_POBLACION/2)):
        c1 = poblacion[np.random.choice(ruleta)]
        c2 = poblacion[np.random.choice(ruleta)]
        hijos = Cromosoma.crossover_parcial(c1, c2)

        for hijo in hijos:
            if np.random.choice([True, False],
p=[K_PROBABILIDAD_MUTACION, 1-K_PROBABILIDAD_MUTACION]):
                Cromosoma.mutacion_intercambio(hijo)

        nueva.extend(hijos)

    return nueva
```

Code 8. Selective pressure definition

```
poblacion = list[Cromosoma]()
nueva_poblacion = list[Cromosoma]()
generacion = 0
```

```
if len(nueva_poblacion) == 0:
    poblacion = [Cromosoma() for _ in range(0, K_POBLACION)]
else:
    poblacion = nueva_poblacion

print('Generation', generacion)
nueva_poblacion = presion_selectiva(poblacion)
generacion += 1
```

Code 9 and 10. Resetting and running code.

```
Generation 0
Best so far:
Combination: [3 1 0 2 6 5 4]
Distances: [ 320 2318 6663 1474 5684 5699 3624]
Evaluation: 25782
```

```
Generation 1
Best so far:
Combination: [3 1 0 2 6 4 5]
Distances: [ 320 2318 6663 1474  697 5699 4817]
Evaluation: 21988
```

```
Generation 3
Best so far:
Combination: [0 1 3 2 6 4 5]
Distances: [2318  320 4065 1474  697 5699 4771]
Evaluation: 19344
```

```
Generation 16
Best so far:
Combination: [0 5 4 6 2 3 1]
Distances: [4771 5699  697 1474 4065  320 2318]
Evaluation: 19344
```

Fig. 6, 7, 8 and 9. Generations of TSP.

One the most interesting things about this is the last image is actually a permutation of the generation 3 result. But, with better operators it surely can work with more cities.

## Conclusions and recommendations

These problems were fun to resolve, even if I struggle with them, it is just like problems in life itself, you became better and you feel comfortable with result, I really find difficult this subject, but, I like how I have been working around it, sometimes I feel a little underestimated about myself, but, deep inside, I feel I still have the courage to keep, and maybe this kind of problems really help with it. I hope my future me become better at resolving problems, I really need something like that.

## References


[1] Wikipedia. Knapsack problem. Aug. 26, 2021. Wikipedia. Accessed on: Oct. 4, 2021. [Online] Available: https://en.wikipedia.org/wiki/Knapsack_problem

[2] Wikipedia. Travelling salesman problem. Oct. 1, 2021. Wikipedia. Accessed on: Oct. 4, 2021. [Online] Available: https://en.wikipedia.org/wiki/Knapsack_problem

[3] Wikipedia. *Genetic algorithm*. Sep. 19, 2021. Wikipedia. Accessed on: Oct. 3, 2021. [Online] Available: https://en.wikipedia.org/wiki/Genetic_algorithm

[4] J. Trigueros, Class lecture, Topic: "Genetic Algorithms" Escuela Superior de Cómputo, Instituto Politécnico Nacional. Mexico City, Sep. 20, 2021.

[5] Infoplease. Air Distances Between World Cities in Statute Miles. Oct. 4, 2021. Infoplease. Accessed on: Oct. 4, 2021. [Online] Available: https://www.infoplease.com/world/travel-transportation/air-distances-between-world-cities-statute-miles

[6] P. Larrañaga, et. al. Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. 1999. Artificial Intelligence Review.