Genetic Algorithms

Luís Santos nº57470

[1] FCUL,
[2] fc57470@alunos.fc.ul.pt

Abstract. Genetic Algorithms are robust and powerful methods that allow to tackle complex optimization and search problems. This project introduces the main building "blocks" that compose a Genetic Algorithm and the underlying parameters.

Keywords: Genetic Algorithms · Optimization · Travelling Salesman Problem.

1 Introduction

The aim of this project was to approach the theme Genetic Algorithms (GAs) at an introductory level by providing a theoretical framework. Alongside implementing a Genetic Algorithm (GA) to solve a simpler version of the Travelling Salesman Problem.

In the first section it will be presented a brief description and conceptualization of the GAs. Furthermore, on the second part of the work it is presented a small section that defines some concepts and terminology used on this area. Afterwards, the third section explains the functionality and the main operators of the GAs. Then, it is concluded with a reflection about the importance of GAs and their parameters, as well, their applications.

2 Conceptualization and Description

The Genetic Algorithms can be defined as search algorithms that function on the mechanics of natural selection and natural genetics [1]. Therefore, like the natural selection and evolution, the processes of genetic algorithms are random. However, they allow to set the level of randomization and control. Making them, more powerful and efficient than random search and brute-force search algorithms [2].

GAs have been developed by the professor John Holland and his team. On his research, Holland aimed to abstract and rigorously explain the adaptive processes of natural systems and to design artificial systems software that applied the mechanisms of natural systems [1]. Belonging to the research area, evolutionary computation, that focuses on designing algorithms for global optimization inspired by the biological evolution [2].

The Genetic Algorithms can be regarded as simple in terms of computation but are powerful in their search for improvement. In addition, they are not fundamentally limited by restrictive assumptions about the search space [1].

The following flowchart is presented to illustrate the different steps that define a GA.

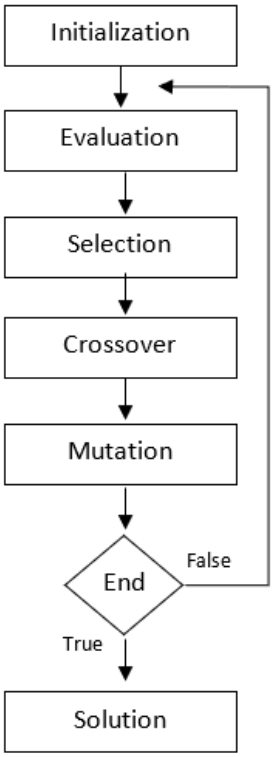

Fig. 1. flowchart of a Genetic Algorithm

3 Terminology

- `Population`: Refers to the set of possible solutions for a given problem [3];
- `Chromosome`: When referring one solution from the population, its is denoted as a chromosome [3];
- `Gene`: The index of each element for a given chromosome is referred as gene [3];
- `Allele`: Is the value that the gene can assume [3];
- `Genotype`: Refers to the representation of the population that is easy to be manipulated and understood using a computing system [3];

- `Phenotype`: It is the representation of the population that refers to the real problem to be solved [3].

4 Travelling Salesman Problem

The travelling salesman problem (TSP) is a combinatorial optimization problem. It is found to have multiple applications in areas such as robotics, transportation, manufacturing, among others [2]. The TSP states that there is a traveling salesman that needs to visit n cities using the shortest route possible. He needs to visit every city exactly once and then returns to the starting point. On the presented project it was defined a simpler version of the TSP, where the cities were represented by 2D points.

5 Main Operators of the Genetic Algorithm

5.1 Initialization

The GA needs to be initialize it, by randomly generating chromosomes that will compose the first population or generation. However, before doing this it is important to consider how the solutions should be represented, in other words, what genotype should be used

There are several ways to represent the genotype of a GA and the type of representation chosen have a great impact on the performance and success of the GA. It must have in account that the representation must define a proper mapping between the phenotype and genotype spaces. Therefore, encoding the genotype and decoding into the phenotype must be done in a correct way [4].

Thus, the most common type of representation is through a binary representation. On this method, the genotype consists of a binary string, it should be used when the solution space consists of boolean decision variables. But it can also be employed to represent numbers on their binary form [4].

But other representations can be used like real value, adequate for a continuous solution space, rather than discrete. Furthermore, it can also be employed integer and permutation representations. The last one should be used when the order of the elements has importance. For TSP this was the correct representation, because the solutions for the problem will be a permutation of all the cities/points that the salesman needs to travel and then return. In addition, with this type of representation it simplifies the computation of the distance for each route [4]. Therefore, each gene will store a number on the range $[0, n]$, being n the number of cities/points defined.
Example of a first population of 5 chromosomes using the permutation representation to solve the TSP, for 6 cities/points:

1. $[0, 3, 2, 4, 1, 5, 0]$;

2. $[0, 4, 1, 3, 2, 5, 0]$;
3. $[0, 1, 5, 4, 3, 2, 0]$;
4. $[0, 3, 5, 4, 2, 1, 0]$;
5. $[0, 2, 3, 5, 1, 4, 0]$.

5.2 Evaluation

Evaluation is key-step that enables the GA to converge to an optimal solution. The evaluation of a population is made through the fitness function. This function tests and quantifies how 'fit' each potential solution is [2]

Therefore, it is the only mechanism that communicates and guides the algorithm to a better solution. An erroneous fitness function will lead to bad solutions, whereas a well-designed function will accomplish the opposite. Nonetheless, the output of a fitness function is not a binary output. It should produce a range of fitness values that accurately scores the chromosomes[2].

On the problem worked the fitness function defined was:

$$F(D_c) = \frac{1}{D_c + 1}, \quad c = 1, ..., n$$

D_c refers to the Euclidean distance computed for each possible chromosome and c refers to the chromosome's number. Through this function, for bigger distances the output will correspond to lower fitness values and for smaller distances it will outputted higher fitness scores.

5.3 Selection

The computing of the fitness of each chromosome is essential in order to proceed with the selection of them. This step focuses on selecting a group of chromosomes to reproduce, passing their genetic information into the offspring, being normally denoted as parents [5].

There are several selection methods that can be employed, like roulette wheel selection or tournament selection. Nonetheless, regarding the technique to be applied it may exert more selective pressure or not. Consequently, it will impact the diversity of the population. Therefore, a high selective pressure will lead to a bigger dominance of the better solutions over the worse ones, leading to a loss of diversity of the population and a premature convergence of the population to a local optimum. On the opposite side, a low selective pressure will increase and preserve the diversity of the population , but it may lead to a slower convergence and result of inefficiency of the evolution process, being similar to a random search [6].

The following selection methods presented are classified as positive selection. However, it can also be employed negative selection, that aims to decides which genotypes are to be removed from the population [6].

Roulette Wheel Selection

This is considered the simplest method of selection, where the individual with higher fitness value has a higher chance of being selected. Thus, this can be visualized as a roulette wheel, where the areas of the wheel are proportion to the fitness value of each individual. So, this roulette wheel 'spin' until the number of individuals selected are reached. Besides this method, there is the stochastic universal sampling, which works similar to the roulette wheel selection. But instead of selecting the individual at each spin, all the chromosomes are selected at one spin. This is achieved by having several pointers at distance of $\frac{F}{n}$, where F corresponds to the total fitness of the population and n corresponds to the total of individuals to be selected. Note that the first pointer is a random value selected from the range $[0, \frac{F}{n}]$ [7].

Tournament Selection

On Tournament Selection k individual are drawn randomly, where k takes values between $[0, n]$ and n is the size of the population. And the winner, the chromosome with the highest fitness value, is selected. The process will be repeated until all spots of the new population are completed [7].

5.4 Crossover

After executing the selection of the chromosomes to reproduce, it will be performed the crossover. The aim of this step is to create new solutions/offspring that inherit the information of the parents [6]. Therefore, crossover is implemented by selecting a random point or points on the parents and exchanging the parts. Resulting from this a new offspring that combines the information of the two parents [8].

On the other hand, the crossover part has the parameter crossover rate. This simply refers to the number of times a crossover occurs for chromosomes in some generation. Therefore, a 100% crossover rate indicates that all chromosomes selected will be picked for crossover and the new generation will be all based on the exchange parts of the parents. On the opposite side, a 0% crossover rate means that any chromosome will be selected for crossover. Thus the new generation will be a exact copy of the older generation [8].

There are several crossover techniques, being the most used the:

- One-point crossover;
- Two-point crossover.

One-point crossover

On the one-point crossover it is selected a random point of the parent chromosomes, and the genes are exchanged. For instances, on the TSP if the parent chromosomes selected for crossover were: $[0, 2, 4, 1, 5, 3, 0]$ and $[0, 2, 3, 1, 5, 4, 0]$, with a random point, $R = 1$. The new offspring resulting from this crossover would be $[0, 2, 3, 1, 5, 4, 0]$ and $[0, 2, 4, 1, 5, 3, 0]$.

Two-point crossover

On this type of crossover two points are generated randomly and are selected on the parent chromosomes for exchange. For example, $[0, 2, 4, 1, 5, 3, 0]$ and $[0, 2, 3, 1, 5, 4, 0]$, with $R = 1$ and $R = 4$, the resulted offspring would be $[0, 2, 3, 1, 5, 3, 0]$ and $[0, 2, 4, 1, 5, 4, 0]$.

5.5 Mutation

Mutation happens after crossover, it applies changes randomly to one or more genes, changing the offspring. Therefore, mutation allows to add diversity to the population, allowing to avoid to be stuck at local optimum [8].

Similar to crossover rate it exists a mutation rate, that determines how many chromosomes should be mutated at each generation. The mutation rate is defined on the interval [0,1] and similar to the crossover rate it prevents into being stuck at the local optima. Nevertheless, high mutation rates may lead to the GA perform as a random search [8].

For instances, taking one of the offspring, $[0, 2, 4, 1, 5, 4, 0]$, created from the crossover, if for the third gene was selected to be mutated, it could change the value for a random value generated, that could be 3. Thus, the final offspring would be $[0, 2, 3, 1, 5, 4, 0]$.

5.6 Other aspects of the Genetic Algorithms

Besides the explained above operations and implied parameters, there is the population's size and number of generations that should also be accountable. The population size is essentially the search space of the algorithm. Thus, determining an adequate population size is no trivial problem. If it is selected a small population's size, means a small search space which will likely lead to converge to some local optimum. On the other hand, a large population size will lead to a more computational load [8].

Furthermore, the number generations which refers to number of cycles before terminating the program. It can be defined the number or it can run until some specific criteria is met [8].

Therefore, it can be defined 4 parameters that need to be adjust and optimize in order to achieve a better or/and fast solution.

- Crossover Rate;
- Mutation Rate;
- Population size;
- Number of generations.

Nevertheless, GAs are stochastic search algorithm. Thus, as showed before GAs apply probabilistic transition rules and not deterministic rules [9]. This may lead to different solutions, even if the all the parameters are all the same, as is showed in the following solutions presented to solve the TSP for 6 cities.

For a crossover rate of 0.3, mutation rate of 0.001, number of generations equal to 100 and an initial population size of 50, this was some of the solutions obtained.
Corresponding Distance: 11.792183669146333
Cities: [array([2., 1.]), array([2.3, 1.2]), array([5. , 1.5]), array([4., 3.]), array([3.4, 4.2]), array([1., 4.]), array([2., 1.])]

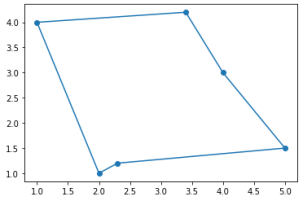

Fig. 2. Solution obtained with GA

Corresponding Distance: 14.13139561274543
Cities: [array([2., 1.]), array([5. , 1.5]), array([3.4, 4.2]), array([4., 3.]), array([1., 4.]), array([2.3, 1.2]), array([2., 1.])]

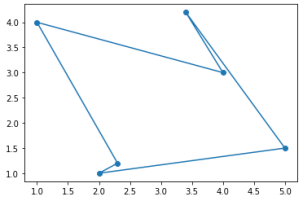

Fig. 3. Solution obtained with GA

Corresponding Distance: 14.579548839709041

Cities: [array([2., 1.]), array([2.3, 1.2]), array([3.4, 4.2]), array([4., 3.]), array([5. , 1.5]), array([1., 4.]), array([2., 1.])]

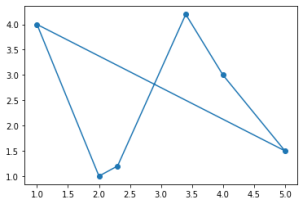

Fig. 4. Solution obtained with GA

Corresponding Distance: 13.348281987128264
Cities: [array([2., 1.]), array([1., 4.]), array([3.4, 4.2]), array([5. , 1.5]), array([4., 3.]), array([2.3, 1.2]), array([2., 1.])]

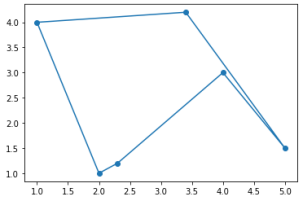

Fig. 5. Solution obtained with GA

Corresponding Distance: 12.041741540772462
Cities: [array([2., 1.]), array([2.3, 1.2]), array([1., 4.]), array([3.4, 4.2]), array([4., 3.]), array([5. , 1.5]), array([2., 1.])]

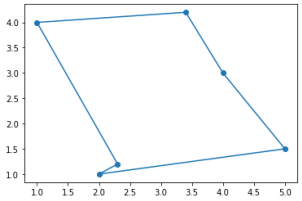

Fig. 6. Solution obtained with GA

From running the algorithm 5 time the GA converge to 5 different solutions. Therefore, the best solution should be choose. On this specific case that corre-

sponds to the solution with a total distance of 11.792183669146333. Not being adequate for the TSP, but for other type of problems it can also be took the average of the several runs of the GA.

6 Conclusion

The aim of this project was to introduce the GAs and how they work, illustrating their operation by applying them to solve the TSP.

GAs have proved to be robust methods for solving optimization and search problems. Being inspired by the natural selection and evolution of species it allowed to develop algorithms that are more than random methods.

This reflects on the wide range of applications that GAs have been employed from designing products that range from large air-crafts to small computer chips to medicines. It was being used in cases for creating realistic special effects for films like Troy and Lord of the Rings. They've been applied to the financial sector on fraud detection and predicting market fluctuations. There are even used for generating new forms of art and music [2]. Further application areas are immune systems, ecology, population genetics, social systems, among others [9].

On the other hand, due to their stochastic nature GAs will tend to produce different results at each run. For such cases, it should be picked the best solution or compute an average of the outputs.

Furthermore, a well-defined fitness function capable of clearly evaluating the fitness of each chromosome, is key to guide the GA to converge to a good solution. Nonetheless, the parameters of the GAs, should also be tuned in order to achieve better results and avoid local optimum.

References

1. Goldberg, D. (2012). *Genetic algorithms in search, optimization, and machine learning.* Boston: Addison-Wesley.
2. Carr, J. (2014). *An Introduction to Genetic Algorithms* [PDF]. Retrieved from https://www.whitman.edu/Documents/Academics/Mathematics/2014/carrjk.pdf
3. Genetic Algorithms - Fundamentals. (2022). Retrieved 1 February 2022, from *https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_fundamentals.htm*
4. Genotype Representation. (2022). Retrieved 1 February 2022, from https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_genotype_representation.htm
5. Shiffman, D. (2012). *The nature of the code.* [S.l.]: D. Shiffman.
6. Komosinski, M. (2021). *Artificial Life and Nature-Inspired Algorithms* [Lecture slides]. Retrieved from http://www.cs.put.poznan.pl/mkomosinski/lectures/MK_ArtLife.pdf
7. Evolutionary Algorithms 3 Selection. (2022). Retrieved 2 February 2022, from http://www.geatbx.com/docu/algindex-02.html#P363_18910

8. Hassanat, A., Almohammadi, K., Alkafaween, E., Abunawas, E., Hammouri, A., & Prasath, V. (2019). Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach. *Information, 10*(12), 390. doi: 10.3390/info10120390
9. Yeh, C. *An Introduction to Genetic Algorithms* [PDF]. Retrieved from https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.496.977&rep=rep1&type=pdf

A Appendix

```
import matplotlib.pyplot as plt
import numpy as np
import random

#Compute the distance between points using the L2 norm
def calcDistance(genes,points):
    dist = 0
    for j in range(len(genes)-1):
        dist+= np.linalg.norm(points[genes[j]] - points[
            genes[j+1]])
    return dist

#Object DNA
class DNA:
    #Constructor
    def __init__(self,lenChromosome,startPoint):
        self.lenChromosome = lenChromosome
        self.startPoint = startPoint
        self.genesArr = []
        self.genesVal = []
        self.genesFitness = []
        self.genesProb = []
        self.selectedGenes = []
        self.childChromosomes = []
        self.totalFitness = None
    #Checks if chrosomes respect the constraints
    def verify(self,genes):
        #First and
        if genes[0] == self.startPoint and genes[-1] ==
            self.startPoint and len(np.unique(genes[0:-1])
            ) == (self.lenChromosome - 1):
            return True
        else:
            return False
    #Instatiate population
    def populate(self,popSize,points):
```

```
    for i in range(0,popSize):
        while True:
            genes = [np.random.randint(0,len(points))
                for i in range(0,dna.lenChromosome)]
            if self.verify(genes):
                self.genesArr.append(genes)
                break
#Fitness function
def fitness(self):
    self.genesFitness.clear()
    for i in range(len(self.genesVal)):
            fitness = 1 / (self.genesVal[i]+1)
            self.genesFitness.append(fitness)
    print("Genes Val:", self.genesVal)

    self.totalFitness = np.sum(self.genesFitness)
    print("Fitness:",self.genesFitness)
    print("Total:",self.totalFitness)

def prob(self):
    self.genesProb = [val/self.totalFitness for val
        in self.genesFitness]
    print("Probabilities:",self.genesProb)
#Selection
def selection(self,n):
    self.selectedGenes.clear()
    rouletteWheel = np.cumsum(self.genesFitness)
    #Stochastic Universal Sampling
    distPointers = self.totalFitness/n
    print(distPointers)
    start = np.random.uniform(0,distPointers)
    #Compute pointers
    pointerArr = [start+i*distPointers for i in range
        (0,n)]
    print(pointerArr)

    #Locate points
    for pointer in pointerArr:
        j=0
        while rouletteWheel[j] < pointer:
            j+=1
        self.selectedGenes.append(self.genesArr[j])
    print("Selected Genes:", self.selectedGenes)
#Crossover
def crossover(self,crossoverRate):
```

```
        crossoverParents = []
        self.childChromosomes.clear()
        #Select Parents
        for i in range(len(self.selectedGenes)):
            r = random.random()
            if r <= crossoverRate:
                crossoverParents.append(self.
                    selectedGenes[i])
            else:
                self.childChromosomes.append(self.
                    selectedGenes[i])

        if crossoverParents:
            #Perform crossover
            for i in range(len(crossoverParents)):
                #If it is the last element
                if (i+1) >= len(crossoverParents):
                    parent1 = crossoverParents[i][1:-1]
                    parent2 = crossoverParents[0][1:-1]
                else:
                    parent1 = crossoverParents[i][1:-1]
                    parent2 = crossoverParents[i+1][1:-1]
                crossoverPoint = np.random.randint(1,self
                    .lenChromosome-1)
                child1 = [self.startPoint] + parent1[0:
                    crossoverPoint] + parent2[
                    crossoverPoint:self.lenChromosome] + [
                    self.startPoint]
                child2 = [self.startPoint] + parent2[0:
                    crossoverPoint] + parent1[
                    crossoverPoint:self.lenChromosome] + [
                    self.startPoint]
                #Verify if the childs are correct
                    according to the rules
                self.childChromosomes.append(child1)

                self.childChromosomes.append(child2)


    #Mutate
    def mutate(self,mutationRate):
        newGen = []
        for chromosome in self.childChromosomes:
            for i in range(1,len(chromosome)-1):
```

```
                    if random.random() < mutationRate:
                        chromosome[i] = np.random.randint(0,
                            dna.lenChromosome-1)

                if self.verify(chromosome):
                    newGen.append(chromosome)
            self.genesArr = newGen
            print("New pop:", self.genesArr)

points = np.array
    ([[2,1],[4,3],[1,4],[5,1.5],[2.3,1.2],[3.4,4.2]])
dna = DNA(len(points)+1,0)
dna.populate(50,points)
print("Init:",dna.genesArr)
print("Decode:",[points[point] for i,point in enumerate(
    dna.genesArr)])

for j in dna.genesArr:
    dna.genesVal.append(calcDistance(j,points))


for i in range(50):
    dna.fitness()
    dna.prob()
    dna.selection(20)
    dna.crossover(0.3)
    dna.mutate(0.001)
    dna.genesVal.clear()
    for j in dna.genesArr:
        dna.genesVal.append(calcDistance(j,points))

#Best solution
solution = [points[point] for i,point in enumerate(dna.
    genesArr[np.argmin(dna.genesVal)])]
print("Best distance: ",dna.genesVal[np.argmin(dna.
    genesVal)])
print("Solution: ",solution)
#Plot route
plt.scatter(*zip(*points))
plt.plot(*zip(*solution))
```