Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Laboratory session #03: Introduction to Genetic Algorithms

Student: Vargas Romero Erick Efrain

Professor: Rosas Trigueros Jorge Luis

Practice completion date: October 27, 2021

Report delivery date: October 08, 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 [2]

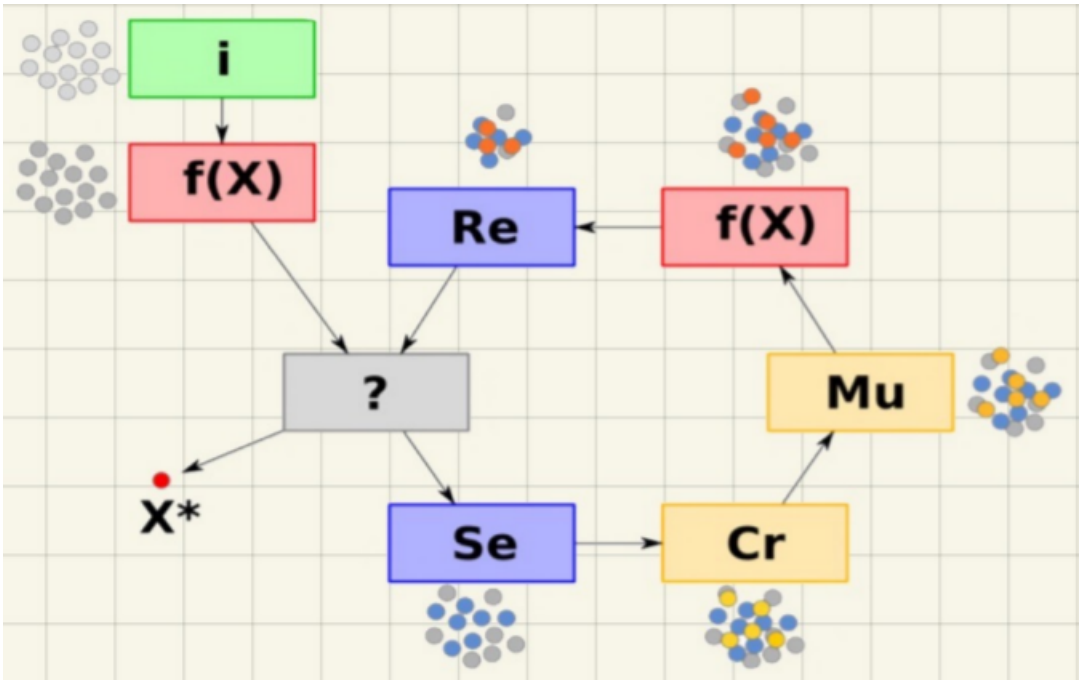

Figure 1.1: Canonical form of genetic algorithms

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 [2].

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

For this practice we will use two particular functions in order to understand how a genetic algorithm works:

1.4 Ackley function

In mathematical optimization the *Ackley function* is a non-convex function used as a performance test for optimization algorithms. It was proposed by *David Ackley* in his 1987 PhD Dissertation [3].

$$f(x,y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5(cos(2\pi x)+cos(2\pi y))} + e + 20 \quad (1.1)$$

1.5 Rastrigin function

In mathematical optimization, the *Rastrigin function* is share some similarities with the *Ackley function*. It is a non-convex function used as a performance test problem for optimization algorithms. It is a typical example of non-linear multimodal function. This function was proposed in 1974 by *Rastrigin* as a 2-dimensional function and has been generalized by *Rudolph* as we can see in the eq. (1.2)[3].

$$f(x_1, x_2, \ldots, x_n) = 10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i)) \quad (1.2)$$

Where where $-5.12 \leq x_i \leq 5.12$ and this function has the global minimum at $f(0,0,\ldots,0) = 0$.

The generalized version was popularized by *Hoffmeister*, *Bäck* and *Mühlenbein*. Finding the minimum of this function is a fairly difficult problem due to its large search space and its large number of local minimal.

For our case we are using the *Rastrigin function* in three dimensions so we get as result the next equation 1.3.

$$f(x,y,z) = 30 + x^2 + y^2 + z^2 - 10cos(2\pi x) - 10cos(2\pi y) - 10cos(2\pi z) \quad (1.3)$$

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:

```
1    python --version
```

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

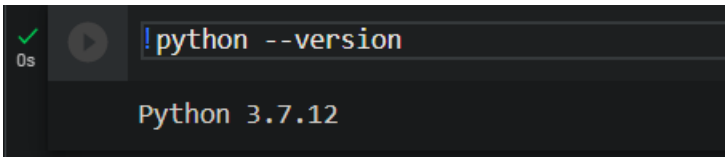

Figure 3.1: Verifying python version

3.1.1 Ackley function

Write a genetic algorithm to minimize the *Ackley's function* in two dimensions.

Based on the file provided in class called *ExampleGA.py* we can try to modify it to add extra dimensions because as we known this example only works for 1-dimension functions right now. So the result of modify this program is the next code:

```
#Aukley's function 2-dimensions

import math
import random
import time
import numpy as np

from functools import cmp_to_key

#Chromosomes
L_chromosome = 8
N_chains=2**L_chromosome
#Lower and upper limits of search space
a = -15
b = 15

#Dimensions
N_dimensions = 2

crossover_point=int(L_chromosome/2)

#Now for each dimension we need a chromosome 'string'
def random_chromosome():
    #Now instead just one dimension we must use two so add an extra for loop could generalize use more than one dimension
    chromosome=[[],[]]
    #Remember than we are using more than one dimension and for each dimension we create a chromosome chain
    for dimension in range(0, N_dimensions):
      for i in range(0,L_chromosome):
          if random.random() < 0.1:
              chromosome[dimension].append(0)
          else:
              chromosome[dimension].append(1)

    return chromosome

#Number of chromosomes
N_chromosomes = 100
#probability of mutation
prob_m=0.5

F0=[]
fitness_values=[]

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

#binary codification
```

```
def decode_chromosome(chromosome):
    global L_chromosome, N_chains, a, b
    #We have two dimensions again so, is mandatory to evaluate each chromosome chain
    value = [0.0, 0.0]

    for dimension in range(0, N_dimensions):
      chain_value = 0.0
      for p in range(L_chromosome):
          chain_value += (2**p) * chromosome[dimension][-1-p]
      value[dimension] = a + (b - a) * float(chain_value) / (N_chains - 1)

    return value


#As we are using two dimensions the function now uses two parameters x and y
def f(x, y):
    return -20 * math.exp(-0.2 * math.sqrt(0.5 * (x**2 + y**2))) - math.exp(0.5 * (
    math.cos(2 * math.pi * x) + math.cos(2 * math.pi * y))) + math.e + 20



def evaluate_chromosomes():
    global F0

    for p in range(N_chromosomes):
        value = decode_chromosome(F0[p])
        fitness_values[p] = f(value[0], value[1])


def compare_chromosomes(chromosome1,chromosome2):
    result_fx1 = decode_chromosome(chromosome1)
    result_fx2 = decode_chromosome(chromosome2)
    value1 = f(result_fx1[0], result_fx1[1])
    value2 = f(result_fx2[0], result_fx2[1])
    if value1 > value2:
        return 1
    elif value1 == value2:
        return 0
    else: #value1 < value2
        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]

    fraction = []
    for p in range(N_chromosomes):
        fraction.append(float(maxv - fitness_values[p]) / acc)
```

```
106         if fraction[-1] <= 1.0 / Lwheel:
107             fraction[-1] = 1.0 / Lwheel
108 
109 ##    print fraction
110     fraction[0] -= (sum(fraction)-1.0)/2
111     fraction[1] -= (sum(fraction)-1.0)/2
112 ##    print fraction
113 
114     wheel=[]
115 
116     pc = 0
117 
118     for f in fraction:
119         Np=int(f*Lwheel)
120         for i in range(Np):
121             wheel.append(pc)
122         pc+=1
123 
124     return wheel
125 
126 F1=F0[:]
127 n=0
128 def nextgeneration():
129     global n
130 
131     F0.sort(key=cmp_to_key(compare_chromosomes) )
132     print( "Best solution so far:")
133     n+=1
134     result = decode_chromosome(F0[0])
135     print(f'{n} f({result}) = {f(result[0], result[1])}')
136 
137     #elitism, the two best chromosomes go directly to the next generation
138     F1[0]=F0[0]
139     F1[1]=F0[1]
140     roulette=create_wheel()
141     for i in range(0,int((N_chromosomes-2)/2)):
142         #Two parents are selected
143         p1=random.choice(roulette)
144         p2=random.choice(roulette)
145         #Two descendants are generated but again we are using two dimensions
146         o1 = [[], []]
147         o2 = [[], []]
148         for dimension in range(0, N_dimensions):
149           o1[dimension] = F0[p1][dimension][0:crossover_point]
150           o1[dimension].extend(F0[p2][dimension][crossover_point:L_chromosome])
151           o2[dimension] = F0[p2][dimension][0:crossover_point]
152           o2[dimension].extend(F0[p1][dimension][crossover_point:L_chromosome])
153           #Each descendant is mutated with probability prob_m
154           if random.random() < prob_m:
155               o1[dimension][int(round(random.random()*(L_chromosome-1)))] ^= 1
156           if random.random() < prob_m:
157               o2[dimension][int(round(random.random()*(L_chromosome-1)))] ^= 1
158         #The descendants are added to F1
159         F1[2+2*i] = o1
160         F1[3+2*i] = o2
161 
162     #The generation replaces the old one
163     F0[:]=F1[:]
```

```
164
165
166 x = list(map(decode_chromosome,F0))
167 y_population = np.zeros(N_chromosomes)
168 F0.sort(key=cmp_to_key(compare_chromosomes))
169 evaluate_chromosomes()
170
171 for i in range(0, 10):
172   nextgeneration()
```

As you can see we added some for loops in functions *random_chromosome*, *decode_chromosome* and *nextgeneration*. As you can see in those for loops mentioned before, we have an extra global variable called *N_dimensions in line* 18. With this extra for loops we can try to generalize as much as possible the program to increase the number of dimensions if you want. Unfortunately is mandatory to modify some other extra line of code to make it work as such as line 25 where we must change from an array to a multidimensional array and we can find other similar cases in the code. Additionally, as you can imagine the function $f(x,y)$ must be changed for the function that you want to use, in this case we used the *Ackley function*.

3.1.2 Rastringin function

Write a genetic algorithm to minimize the *Rastrigin's function* in two dimensions.

To this problem we can re-use the previous code of the *Ackley function*, because we tried to generalize the use of extra dimensions but unfortunately we must change some lines of code. The result of all the modifications is the next code:

```
#Rastringin's function 3-dimensions

import math
import random
import time
import numpy as np

from functools import cmp_to_key

#Chromosomes
L_chromosome = 8
N_chains=2**L_chromosome
#Lower and upper limits of search space
a = -10
b = 10
#Dimensions
N_dimensions = 3

crossover_point=int(L_chromosome/2)

#Now for each dimension we need a chromosome 'string'
def random_chromosome():
    #Now instead just one dimension we must use three so add an extra for loop could
     generalize use more than one dimension
    chromosome=[[],[],[]]
    #Remember than we are using more than one dimension and for each dimension we
    create a chromosome chain
    for dimension in range(0, N_dimensions):
      for i in range(0, L_chromosome):
          if random.random() < 0.5:
              chromosome[dimension].append(0)
          else:
              chromosome[dimension].append(1)

    return chromosome

#Number of chromosomes
N_chromosomes = 300
#probability of mutation
prob_m = 0.5

F0=[]
fitness_values=[]

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

#binary codification
def decode_chromosome(chromosome):
```

```
49      global L_chromosome,N_chains,a,b
50      #We have three dimensions again so, is mandatory to evaluate each chromosome
        chain
51      value = [0.0, 0.0, 0.0]
52
53      for dimension in range(0, N_dimensions):
54        chain_value = 0.0
55        for p in range(L_chromosome):
56            chain_value += (2**p) * chromosome[dimension][-1-p]
57        value[dimension] = a + (b - a) * float(chain_value) / (N_chains - 1)
58
59      return value
60
61
62  #As we are using two dimensions the function now uses two parameters x and y
63  def f(x, y, z):
64      return 30 + x**2 + y**2 + z**2 - 10 * math.cos(2 * math.pi * x) - 10 * math.cos
        (2 * math.pi * y) - 10 * math.cos(2 * math.pi * z)
65
66  def evaluate_chromosomes():
67      global F0
68
69      for p in range(N_chromosomes):
70          value = decode_chromosome(F0[p])
71          fitness_values[p] = f(value[0], value[1], value[2])
72
73
74  def compare_chromosomes(chromosome1,chromosome2):
75      result_fx1 = decode_chromosome(chromosome1)
76      result_fx2 = decode_chromosome(chromosome2)
77
78      value1 = f(result_fx1[0], result_fx1[1], result_fx1[2])
79      value2 = f(result_fx2[0], result_fx2[1], result_fx2[2])
80      if value1 > value2:
81          return 1
82      elif value1 == value2:
83          return 0
84      else: #value1 < value2
85          return -1
86
87
88  suma = float(N_chromosomes*(N_chromosomes+1))/2.
89
90  Lwheel = N_chromosomes*10
91
92  def create_wheel():
93      global F0,fitness_values
94
95      maxv=max(fitness_values)
96      acc=0
97      for p in range(N_chromosomes):
98          acc+=maxv-fitness_values[p]
99      fraction=[]
100     for p in range(N_chromosomes):
101         fraction.append( float(maxv-fitness_values[p])/acc)
102         if fraction[-1]<=1.0/Lwheel:
103             fraction[-1]=1.0/Lwheel
104 ##    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():
    global n

    F0.sort(key=cmp_to_key(compare_chromosomes) )
    print( "Best solution so far:")
    n+=1
    result = decode_chromosome(F0[0])
    print(f'{n} f({result}) = {f(result[0], result[1], result[2])}')

    #elitism, the two best chromosomes go directly to the next generation
    F1[0]=F0[0]
    F1[1]=F0[1]
    roulette=create_wheel()
    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 but again we are using three dimensions
        o1 = [[], [], []]
        o2 = [[], [], []]
        for dimension in range(0, N_dimensions):
          o1[dimension] = F0[p1][dimension][0:crossover_point]
          o1[dimension].extend(F0[p2][dimension][crossover_point:L_chromosome])
          o2[dimension] = F0[p2][dimension][0:crossover_point]
          o2[dimension].extend(F0[p1][dimension][crossover_point:L_chromosome])
          #Each descendant is mutated with probability prob_m
          if random.random() < prob_m:
              o1[dimension][int(round(random.random()*(L_chromosome-1)))] ^= 1
          if random.random() < prob_m:
              o2[dimension][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[:]


x = list(map(decode_chromosome,F0))
y_population = np.zeros(N_chromosomes)
```

```
163 F0.sort(key=cmp_to_key(compare_chromosomes))
164 evaluate_chromosomes()
165
166 for i in range(0, 100):
167   nextgeneration()
```

In this code as you can see now the all the variables called *chromosome* are 3-dimensional arrays we can see this in line 24. Obviouslly we must change the content of the function $f(x,y)$ because we are using an extra dimension so instead of receive two parameters now we have three $f(x,y,z)$ and the content is the *Rastringin function* in its 3-dimensional version as we saw in the eq. (1.3)

4 Screens, graphs and diagrams

4.1 Ackley function

For this function we used a lower value of -15 and an upper value of 15 in order to make easier to find the optimal solution, additionally we increased the number of chromosomes to 100. We ran the program and with 10 iterations we found values very near of the global minimum value, in our case we found the point $(-0.0588235294117645, -0.0588235294117645) = 0.4114144574357006$, you can see more information in the fig. 4.1

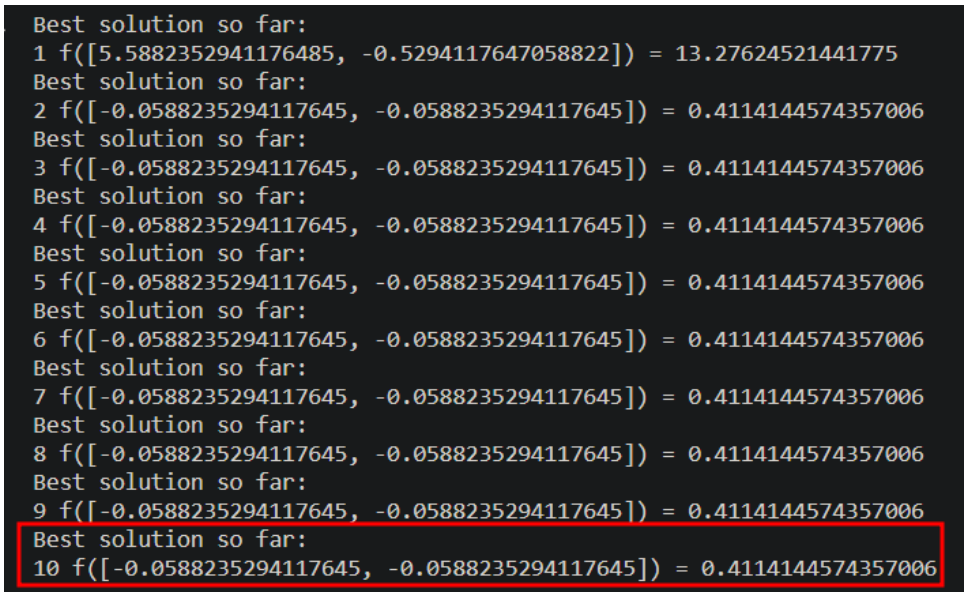

Figure 4.1: Ackley function simulation

4.2 Rastrigin function

In the other hand we have the *Rastrigin function* case, here we used a lower value of -10 and an upper value of 10. Again we increased the number of chromosomes to 300 in order to help the genetic algorithm to find the optimal solution. We ran the program and with 100 iterations we found values very near of the global minimum value you can appreciate the result in fig. 4.2. Additionally if you ran several times the algorithm or if you increase the number of iteratons you would find a value so much near of point $(0,0,0)$ and it is the global minimum of this function.

```
83 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
84 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
85 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
86 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
87 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
88 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
89 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
90 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
91 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
92 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
93 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
94 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
95 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
96 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
97 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
98 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
99 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
Best solution so far:
100 f([0.039215686274509665, 0.9803921568627452, 0.039215686274509665]) = 1.6441007039183955
```

Figure 4.2: Rastrigin function simulation

5 Conclusions

In this practice we were able to see the differences between three fitness functions (if you take in count the original file it is the third fitness function). First of all, we understand how the original file works in order to be able to modify the program and increase the number of dimensions. Then "play" with the code helps to discover interesting things like what happen if we increase the number of cromosomes or if we change the lower and upper values (the limits of the search space) change those values or variables can help us to get a better solution faster. For example in the case of the *Rastrigin function* if you do not reduce the search space you would need more iterations to get the optimal solution and something else that you need take in count is the number of cromosomes this is super important too, because you have more information and it can help the algorithm to find the solution faster.

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] U. Bondenhofer. *Genetic Algorithms: Theory and Applications*. Third Edition. Accessed on October 25th, 2021. Available online:http://www.flll.jku.at/div/teaching/Ga/GA-Notes.pdf

[3] L. Idoumghar, R. Schott and M. Melkemi. *A NOVEL HYBRID EVOLUTIONARY ALGORITHM FOR MULTI-MODAL FUNCTION OPTIMIZATION AND ENGINEERING APPLICATIONS*. September 2009. Accessed on October 25th, 2021. Available online: https://www.researchgate.net/publication/235950622_A_novel_hybrid_evolutionary_algorithm_for_multi-modal_function_optimization_and_engineering_applications/download