Instituto Politécnico Nacional

ESC Superior de Cómputo

"EVOLUTIONARY COMPUTING" 2-ACKLEY AND RASTRIGIN PROBLEMS

Jorge Luis Rosas Trigueros

Saldaña Aguilar Gabriela

18/09/16 - 23/09/16

THEORETICAL FRAMEWORK

Evolutionary algorithms have been shown to be successful for a wide range of optimization problems. While these randomized search heuristics work well for many optimization problems in practice, a satisfying and rigorous mathematical understanding of their performance is an important challenge in the area of genetic and evolutionary computing. It has been proven for various combinatorial optimization problems that they can be solved by evolutionary algorithms in reasonable time using a suitable representation together with mutation operators adjusted to the given problem. The representations used in these papers are different from the general encodings working with binary strings as considered earlier in theoretical works on the runtime behavior of evolutionary algorithms. the chosen representations reflect some properties of partial solutions of the problem at hand that allow to obtain solutions that can be extended to optimal ones for the considered problem.

Dynamic programming is a well-known algorithmic technique that helps to tackle a wide range of problems. A general framework for dynamic programming has been considered by e. g. Woeginger and Kogan . The technique allows the design of efficient algorithms, that solve the problem at hand to optimality, by extending partial solutions to optimal ones.

Framework for Evolutionary Algorithms An evolutionary algorithm consists of different generic modules, which have to be made precise by the user to best fit to the problem. Experimental practice, but also some theoretical work, demonstrate that the right choice of representation, variation operators, and selection method is crucial for the success of such algorithms. We assume that the problem to be solved is given by a multi-objective function g that has to be optimized. We consider simple evolutionary algorithms that consist of the following components. The algorithm (see Algorithm 2) starts with an initial population P0.

During the optimization the evolutionary algorithm uses a selection operator sel(·) and a mutation operator mut(·) to create new individuals. The d-dimensional fitness function together with a partial order par on R d induce a partial order dom on the phenotype space, which guides the search. After the termination of the EA, an output function out(·) is utilized to map the individuals in the last population to search points from the original search space.

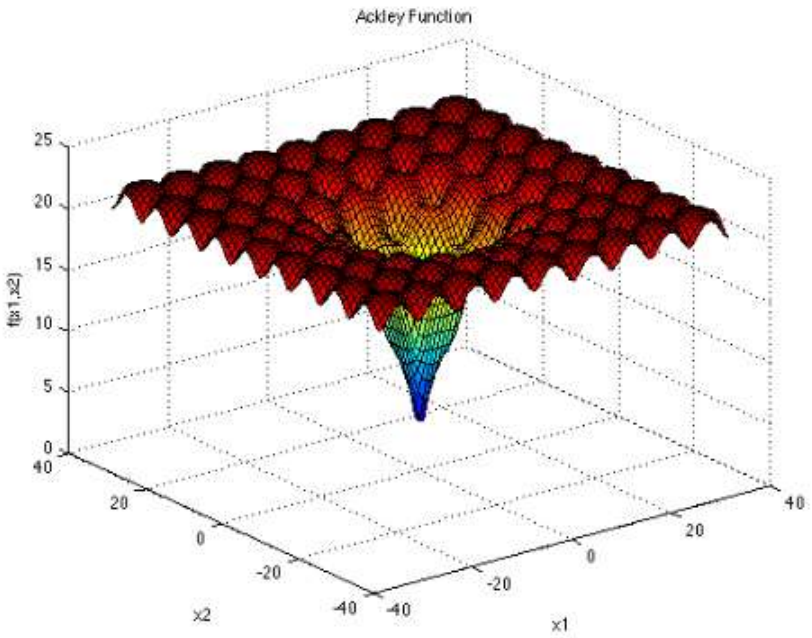
TEST FUNCTIONS FOR OPTIMIZATION

In applied mathematics, **test functions**, known as **artificial landscapes**, are useful to evaluate characteristics of optimization algorithms, such as:

- Velocity of convergence.
- Precision.
- Robustness.
- General performance.

Here some test functions are presented with the aim of giving an idea about the different situations that optimization algorithms have to face when coping with these kinds of problems. In the first part, some objective functions for single-objective optimization cases are presented

ACKLEY FUNCTION



$$f(\mathbf{x}) = -a\exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d}x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right) + a + \exp(1)$$

Description:
Dimensions: d

The Ackley function is widely used for testing optimization algorithms. In its two-dimensional form, as shown in the plot above, it is characterized by a nearly flat outer region, and a large hole at the centre. The function poses a risk for optimization algorithms, particularly hillclimbing algorithms, to be trapped in one of its many local minima.

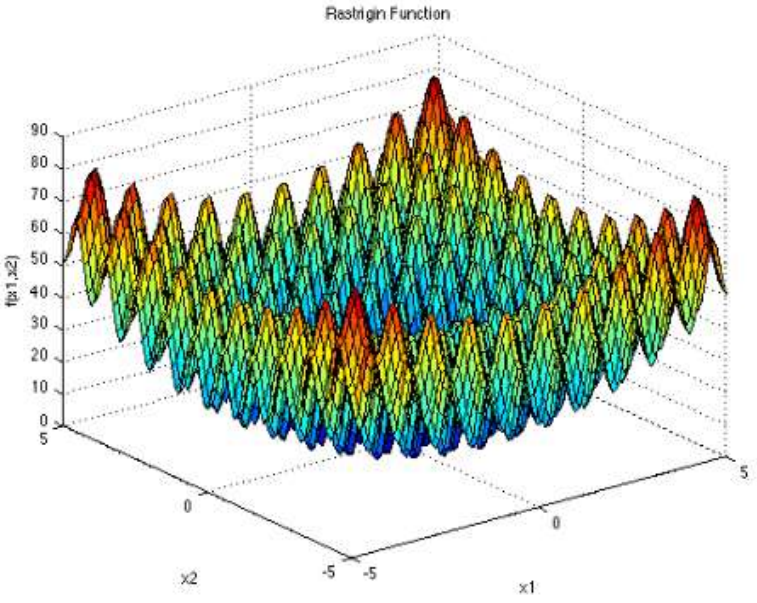
Recommended variable values are: a = 20, b = 0.2 and c = 2π.

Input Domain:
The function is usually evaluated on the hypercube $x_i \in [-32.768, 32.768]$, for all i = 1, …, d, although it may also be restricted to a smaller domain.

Global Minimum:

$$f(\mathbf{x}^*) = 0, \text{ at } \mathbf{x}^* = (0, \ldots, 0)$$

RASTRIGIN FUNCTION



$$f(\mathbf{x}) = 10d + \sum_{i=1}^{d} \left[x_i^2 - 10\cos(2\pi x_i) \right]$$

Description:
Dimensions: d

The Rastrigin function has several local minima. It is highly multimodal, but locations of the minima are regularly distributed. It is shown in the plot above in its two-dimensional form.

Input Domain:
The function is usually evaluated on the hypercube $x_i \in [-5.12, 5.12]$, for all i = 1, …, d.

Global Minimum:
$f(\mathbf{x}^*) = 0$, at $\mathbf{x}^* = (0, \ldots, 0)$

EQUIPMENT AND MATERIAL.

SOFTWARE: The program is made in Python

BODY

We were told to write down the changes to the code provided by the teacher that simulates new generations every time we click evolve.

The first issue I found is how to implement both functions (Ackley an Ratrigin) using two variables, because I didn' t want to use a lot of code in that section, fortunately, with a little help of my friends we find out how this works and what do we need to modify. The final result `(see e. g. [1, 2]).`

`F1: Ackley's`

Mutation probability	Chromosome length: 4	8	16
0.1	100 generations. f(-3.072,3.072)=20.8862629687	100 generations. f(-4.959, 1.947)= 29.2495540763	100 generations. f(-5.080, 1.000)= 28.0498184393
0.25	100 generations. f(-3.072,3.072)=20.8862629687	100 generations. f(-4.999, -1.907))= 30.27732506	100 generations f((-5.080, 1.000)= 28.0498184393
0.5	100 generations. f(-3.072,3.072)=20.8862629687	100 generations. f(-4.959, -1.264)=37.456903701	100 generations. f(-5.080, -0.039)= 27.3641820461
0.75	100 generations. f(-3.072,3.072)=20.8862629687	100 generations. f(-4.959, 1.947)= 29.249554076	100 generations. f(-5.080, -0.039)= 27.3641820461

F2: Rastrigin's

Chromosome length

Mutation probability

	4	8	16
0.1	100 generations. f(-3.072, 0.3413)= 30.4879375	100 generations. f(-4.517, -0.622)= 34.25411666	100 generations. f(-5.080, -0.039)= 35.1149453458
0.25	100 generations. f-3.072, 0.3413)= 30.48793754	100 generations. f(-4.517, 0.020)= 33.999723309	f(-5.080, 7.812)= 35.1137156251
0.5	100 generations. f(-3.072, 0.341)= 30.48793754	100 generations. f(-4.517, -0.622)= 34.25411666	100 generations. f(-5.080, -0.039)= 35.1149453458
0.75	100 generations. f(-3.072, 0.341)= 30.487937546	100 generations. f((-4.517, 0.020)= 33.99972330	100 generations. f(-5.080, -0.039)= 35.1149453458

CODE

```
#Ackley
from Tkinter import *
import math
import random
#Chromosomes are 4 bits long
#Como no hay una cadena que sea extrictamente cero nunca llega a cero
#REGISTRAR EL VALOR con 4 bits
L_chromosome=16
N_chains= 2**L_chromosome #poblacion
#Lower and upper limits of search space

#espacio de busqueda
a=-5.12
b=5.12
crossover_point=L_chromosome/2#punto de cruce

#cadena con dos variables arreglo mitad x y mitad y
def random_chromosome():#generamos un cromosoma
    chromosome=[]
    for i in range(0,L_chromosome):
        if random.random()<0.1:
            chromosome.append(0)
        else:
            chromosome.append(1)

    return chromosome

#Number of chromosomes
N_chromosomes=10
#probability of mutation
#prob_m=0.1
#prob_m=0.25
#prob_m=0.5
prob_m=0.75

F0=[]#poblacion original, antes de ser mutada
fitness_values=[]#valores de aptitud de cada inidividuo

#generamos los individuos de la poblacion original
for i in range(0,N_chromosomes): #array de cromosomas
    F0.append(random_chromosome())
    fitness_values.append(0)#llenamos de ceros

#binary codification-->FENOTIPO
def decode_chromosome(chromosome):
    global L_chromosome,N_chains,a,b
    value=0
    value2=0
    for p in range(0,L_chromosome/2):
        value+=(2**p)*chromosome[-1-p]

    for p in range(L_chromosome/2, L_chromosome):
        value2+=(2**p)*chromosome[-1-p]

    return (a+(b-a)*float(value)/(N_chains-1),a+(b-a)*float(value2)/(N_chains-1))

#funcion de aptitud, es una parabola, genera muchos max y min locales
#modificar enviar dos parametros f(x,y)
def f(x,y):
    #return 0.05*x*x-4*math.cos(x)
    return ((20.0) + ((x**2)- (10.0*(math.cos((2.0*math.pi)*x)))) + ((y**2) - (10.0*(math.cos((2.0*math.pi)* y)))))
```

```
#evaluamos su funcion de aptitud
def evaluate_chromosomes():
    global F0
    for p in range(N_chromosomes):
        (v,v1)=decode_chromosome(F0[p])
        fitness_values[p]=f(v,v1)

#funcion de comparacion con sort, definimos como se
#van a comparar , se comparan las funciones de los
#cromosomas
def compare_chromosomes(chromosome1,chromosome2):
    (vx1,vy1)=decode_chromosome(chromosome1)
    (vx2,vy2)=decode_chromosome(chromosome2)
    fvc1=f(vx1,vy1)
    fvc2=f(vx2,vy2)
    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

#El mas alto va a tener mas boletos en la ruleta
#y el menos menos boletos
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)
        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[:]
```

```
#Evaluamos, ordenamos la poblacion inicial f0
#CON EL ELITISMO GARANTIZAMOS de que llegue al optimo
#Se genera la ruleta y se sacan dos individuos que
#van a ser los progenitores
#Luego damos un valor alto de probabilidad de mutacion
#se le hace un XOR para mutar y se agregan a la poblacion de
#descendencia
n_generation=0
def nextgeneration():
    global n_generation
    n_generation+=1
    print n_generation

    F0.sort(cmp=compare_chromosomes)
    print "Best solution so far:"
    (x,y)=decode_chromosome(F0[0])
    print "f(",decode_chromosome(F0[0]),")= ", f(x,y)

    #elitism, the two best chromosomes go directly to the next generation
    F1[0]=F0[0]
    F1[1]=F0[1]

    for i in range(0,(N_chromosomes-2)/2):
        roulette=create_wheel()
        #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[:]

F0.sort(cmp=compare_chromosomes)
evaluate_chromosomes()

for i in range(1,100):

            nextgeneration()
```

```
#Rastrigin

from Tkinter import *
import math
import random
#Chromosomes are 4 bits long
#Como no hay una cadena que sea extrictamente cero nunca llega a cero
#REGISTRAR EL VALOR con 4 bits
L_chromosome=16
N_chains= 2**L_chromosome #poblacion
#Lower and upper limits of search space

#espacio de busqueda
a=-5.12
b=5.12
crossover_point=L_chromosome/2#punto de cruce

#cadena con dos variables arreglo mitad x y mitad y
def random_chromosome():#generamos un cromosoma
    chromosome=[]
    for i in range(0,L_chromosome):
        if random.random()<0.1:
            chromosome.append(0)
        else:
            chromosome.append(1)

    return chromosome

#Number of chromosomes
N_chromosomes=10
#probability of mutation
#prob_m=0.1
#prob_m=0.25
#prob_m=0.5
prob_m=0.75

F0=[]#poblacion original, antes de ser mutada
fitness_values=[]#valores de aptitud de cada inidividuo

#generamos los individuos de la poblacion original
for i in range(0,N_chromosomes): #array de cromosomas
    F0.append(random_chromosome())
    fitness_values.append(0)#llenamos de ceros

#binary codification-->FENOTIPO
def decode_chromosome(chromosome):
    global L_chromosome,N_chains,a,b
    value=0
    value2=0
    for p in range(0,L_chromosome/2):
        value+=(2**p)*chromosome[-1-p]

    for p in range(L_chromosome/2, L_chromosome):
        value2+=(2**p)*chromosome[-1-p]

    return (a+(b-a)*float(value)/(N_chains-1),a+(b-a)*float(value2)/(N_chains-1))
```

```
#funcion de aptitud, es una parabola, genera muchos max y min locales
#modificar enviar dos parametros f(x,y)
def f(x,y):
    #return 0.05*x*x-4*math.cos(x)
    a=20.0
    b=0.2
    c=(2.0*math.pi)
    return ((-a)*((-b)*((((x**2)*(0.5)) + ((y**2) *(0.5)))**(0.5))))-(math.exp(((math.cos(c+x))*(0.5)) +((math.cos(c+y))*(0.5)))) +
a + math.exp(1.0)

#evaluamos su funcion de aptitud
def evaluate_chromosomes():
    global F0
    for p in range(N_chromosomes):
        (v,v1)=decode_chromosome(F0[p])
        fitness_values[p]=f(v,v1)

#funcion de comparacion con sort, definimos como se
#van a comparar , se comparan las funciones de los
#cromosomas
def compare_chromosomes(chromosome1,chromosome2):
    (vx1,vy1)=decode_chromosome(chromosome1)
    (vx2,vy2)=decode_chromosome(chromosome2)
    fvc1=f(vx1,vy1)
    fvc2=f(vx2,vy2)
    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

#El mas alto va a tener mas boletos en la ruleta
#y el menos menos boletos
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)
        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[:]

#Evaluamos, ordenamos la poblacion inicial f0
#CON EL ELITISMO GARANTIZAMOS de que llegue al optimo
#Se genera la ruleta y se sacan dos individuos que
#van a ser los progenitores
#Luego damos un valor alto de probabilidad de mutacion
#se le hace un XOR para mutar y se agregan a la poblacion de
#descendencia
n_generation=0
def nextgeneration():
    global n_generation
    n_generation+=1
    print n_generation

    F0.sort(cmp=compare_chromosomes)
    print "Best solution so far:"
    (x,y)=decode_chromosome(F0[0])
    print "f(",decode_chromosome(F0[0]),")= ", f(x,y)
    #elitism, the two best chromosomes go directly to the next generation
    F1[0]=F0[0]
    F1[1]=F0[1]

    for i in range(0,(N_chromosomes-2)/2):
        roulette=create_wheel()
        #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[:]
F0.sort(cmp=compare_chromosomes)
evaluate_chromosomes()

for i in range(1,100):

    nextgeneration()
```

CONCLUSION

I liked it the way we took the class in the lab like if it was free style, but with the help of the professor, also it is good to explain what will be the next topic to cover in the lab so the students can find it out and investigate a little bit before entering the class because it could be confusing if it is our first time coding evolutionary algorithms.

BIBLIOGRAPHY

NAME: http://www.sfu.ca/~ssurjano/rastr.html

DATE: 07/01/2015

NAME: http://www.sfu.ca/~ssurjano/ackley.html

DATE: 07/01/2015