26 de Septiembre 2020

Optimización II

Reporte Práctica 1

Algoritmos Genéticos

Alfonso Giuseppe Carte Granillo

2163071549

Implementación

Para el desarrollo de este algoritmo genético ocupé el lenguaje de programación Python, a continuación presentaré secuencialmente las partes del código con una breve explicación de lo que hace cada una de esas partes, después presentaré las gráficas de los diferentes resultados ocupando 128bits, 256bits, 512bits y por último 1024bits, el código es posible encontrarlo en el siguiente link: https://github.com/Giuseppecarte/Genetic_algorithm

Desarrollo del algoritmo:

Librerías:

Las librerías que ocupé fueron:

- Numpy: para el manejo de arreglos y la creación de números aleatorios
- Tensorflow: Para usar la función de shuffle la cual, barajea de forma aleatoria los elementos de una lista
- Pandas: Para crear las tablas donde se guardarán los resultados de las diferentes estrategias
- Matplotlib: Para graficar los resultados de las diferentes estrategias

```
import numpy as np
from tensorflow.random import shuffle
import pandas as pd
import matplotlib.pyplot as plt
```

Datos iniciales:
En esta parte definimos las variables globales las cuales ocuparemos durante todo el algoritmo, así como la población inicial aleatoria la cual tiene como nombre *Pinic* al igual que el vector *gamma*

```
prob_cruza = 0.9
tam_pob = 100
bits = 256
prob_mut = 1/bits
G_max = 100
gamma = [0 if i%2 == 0 else 1 for i in range(bits)]
#Población 0
Pinic = [list(np.random.randint(0,2,bits)) for i in range(tam_pob)]
```

Funciones para la estructura general del algoritmo:

Distancia de Hamming:
Esta es la función objetivo, la cual deseamos minimizar.

```
def Hamming(genotipo, gamma = gamma):
    assert len(genotipo)==len(gamma)
    peso = 0
    for j in range(len(genotipo)):
        if genotipo[j] == gamma[j]:
            peso += 1
        else: pass
    return peso
```

Selección de padres:

En esta función suceden dos cosas importantes, la primera es que las variables *P1* y *P2* son los dos barajeos de la población inicial *Pinic*, y la otra cosa es la función pelea_padres, la cual selecciona de dos en dos los padres que obtengan una distancia de *Hamming* menor.

```
def padres(Pinic):
    #Hacemos las mezclas
    P1 = shuffle(Pinic).numpy().tolist()
    P2 = shuffle(P1).numpy().tolist()
    #parte 2
    def pelea_padres(padres):
        padres_ganadores = []
        for j in list(range(0,len(padres),2)):
            if Hamming(padres[j]) <= Hamming(padres[j+1]):
                padres_ganadores.append(padres[j])
            else: padres_ganadores.append(padres[j+1])
        return padres_ganadores
    p1_prima = pelea_padres(P1)
    p2_prima = pelea_padres(P2)
    return p1_prima, p2_prima
```

Cruce:

En esta función sucede el cruce entre los padres que resultaron ganadores de la selección de padres, los cuales tienen un genotipo ‘mejor’ que los padres que no pasaron de esa selección, al final esta función nos devuelve una lista *Q* la cual contiene ya sean los hijios resultante de la cruza de los padres o en caso que el número aleatorio fuera mayor que la probabilidad de cruza, añadiría en lugar de los hijos a esos mismos padres.

```
def cruza(padres1, padres2):
    assert len(padres1)==len(padres2)
    Q = []
    for i in range(len(padres1)):
        num_aleatorio = np.random.random(1)[0]
        #Aquí
        if num_aleatorio <= prob_cruza:
            aleatorio = np.random.randint(1, len(padres1))
            x = padres1[i]
            y = padres2[i]
            resto = abs(aleatorio - len(x))

            h1 = [*x[:aleatorio],*y[-resto:]]
            h2 = [*x[-resto:],*y[:aleatorio]]
            # original
            # h2 = [*y[:aleatorio],*x[-resto:]]
            Q.append(h1)
            Q.append(h2)
        else:
            Q.append(padres1[i])
            Q.append(padres2[i])
    return Q
```

Mutación:

En esta función lo que sucede es que dependiendo del número aleatorio generado y la probabilidad de mutación *Pm* los genotipos mutan es decir en los genes de un genotipo los cuales fueran 0's cambian a 1's y viceversa.

```
def mutacion(Q):
    Q_prima = []
    for genotipo in Q:
        num_aleatorio = np.random.random(1)[0]
        #Aquí!!!!!!
        if num_aleatorio <= prob_mut:
            genotipo_prima = [0 if j==1 else 1 for j in genotipo ]
            Q_prima.append(genotipo_prima)
        else: Q_prima.append(genotipo)
    return Q_prima
```

Diferentes estrategias:

En esta sección presentaremos las funciones de las siguientes 3 estrategias:

- $(\mu + \lambda)$
- $(\mu \lambda)$
- $(\mu \lambda)\ con\ elitismo$

Estrategia $(\mu + \lambda)$:

En esta estrategia lo que sucede es que concatenamos las *Pinic* con el resultado de *Q*, las evaluamos con la distancia de *Hamming* y luego las colocamos de menor a mayor según las distancia de *Hamming*, ya que las tenemos ordenadas partimos esa lista resultante a la mitad y esa lista resultante la definimos como la siguiente *Pinic* e iteramos hasta que se llegue al máximo de generaciones *G_max.*

La tabla llamada *datos_finales*, contiene a los mejores genotipos junto con su fenotipo de cada generación.

```python
def mu_mas_l(Pinic=Pinic):
    conteo = 0
    result = []
    genotipos = []
    generaciones = []
    while conteo < G_max:
        p1_prima, p2_prima = padres(Pinic)
        Q = cruza(p1_prima, p2_prima)
        concat = Pinic + Q
        eval_concat = [Hamming(line) for line in concat]
        datos = pd.DataFrame({'Genotipo':[line for line in concat], 'F(x)':eval_concat}).sort_values('F(x)')
        datos = datos.iloc[:int(len(concat)/2)]
        #print(f"Generación:{conteo} : {datos['Genotipo'].iloc[0]} : {datos['F(x)'].iloc[0]}")
        generaciones.append('Generación : '+str(conteo))
        genotipos.append(datos['Genotipo'].iloc[0])
        result.append(datos['F(x)'].iloc[0])
        Pinic = list(datos['Genotipo'])
        conteo += 1
    datos_finales = pd.DataFrame({'Generaciones':generaciones, 'Genotipos':genotipos, 'F(x)':result})
    return datos_finales
```

Estrategia $(\mu \lambda)$

En esta estrategia lo que hacemos es que iteramos hasta que se llegue el máximo de generaciones y dentro de estas iteraciones hacemos que la *Pinic* la cambiamos con el resultado de la función *mutación* el cual es la lista llamada *Q_prima.* Sin olvidar que dentro de cada iteración los ordenamos de menor distancia de *Hamming* a mayor con el fin de sacar al mejor genotipo durante cada iteración.

```python
def m_l(Pinic = Pinic):
    conteo = 0
    result = []
    genotipos = []
    generaciones = []
    while conteo < G_max:
        p1_prima, p2_prima = padres(Pinic)
        Q_prima = mutacion(cruza(p1_prima, p2_prima))
        datos_1 = pd.DataFrame({'Genotipo':[genotipo for genotipo in Q_prima], 'F(x)':[Hamming(genotipo) for genotipo in Q_prima]}).sort_values('F(x)')
        #print(f"Generación:{conteo} : {datos_1['Genotipo'].iloc[0]} : {datos_1['F(x)'].iloc[0]}")
        generaciones.append('Generación : '+str(conteo))
        genotipos.append(datos_1['Genotipo'].iloc[0])
        result.append(datos_1['F(x)'].iloc[0])
        Pinic = list(datos_1['Genotipo'])
        conteo += 1
    datos_finales = pd.DataFrame({'Generaciones':generaciones, 'Genotipos':genotipos, 'F(x)':result})
    return datos_finales
```

Estrategia $(\mu\lambda)\ con\ elitismo$

Por último desarrollé la estrategia con elitismo, la cual tiene como particular que hace la evaluación a cada genotipo con la distancia de *Hamming* de la Población inicial *Pinic* y los acomoda de menor a mayor. A su vez acomoda de la misma forma a los genotipos resultantes de la función mutación.

Después de que tenemos acomodados tanto a los elementos de *Pinic* y los de *Q'* , sacamos al mejor genotipo de *Pinic* y al pero genotipo de *Q'*, definimos a *Q"* la cual es la unión de *Q* sin su peor genotipo y al mejor genotipo de *Pinic*. Acomodamos de nuevo según la distancia de *Hamming* a la nueva lista *Q"* e iteramos cambiando *Pinic* con Q" en cada evaluación.

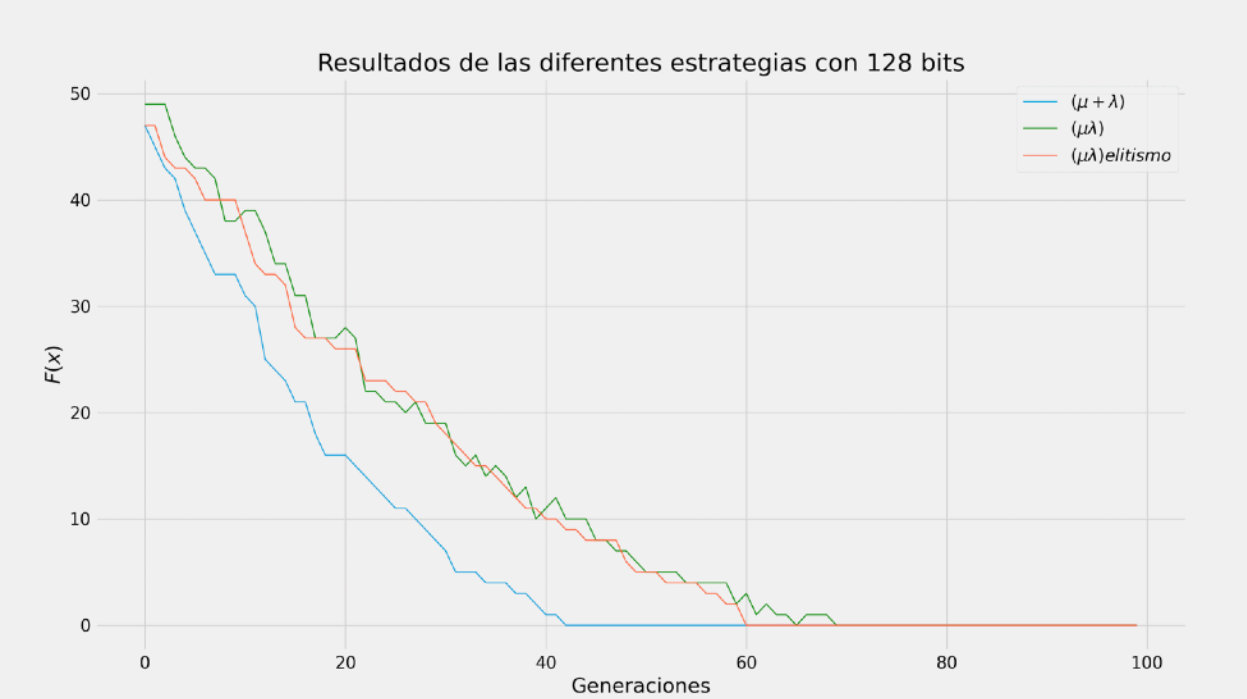
```python
def mu_l_elitismo(Pinic = Pinic):
    conteo = 0
    result = []
    genotipos = []
    generaciones = []
    while conteo < G_max:
        p1_prima, p2_prima = padres(Pinic)
        mejor_Pinic = [Hamming(genotipo) for genotipo in Pinic]
        datos_2 = pd.DataFrame({'Genotipo':[gen for gen in Pinic], 'F(x)':mejor_Pinic})
        #Sacamos el mejor de Pinic

        Q_prima = mutacion(cruza(p1_prima, p2_prima))
        datos_1= pd.DataFrame({'Genotipo':[genotipo for genotipo in Q_prima], 'F(x)':[Hamming(genotipo) for genotipo in Q_prima]}).sort_values('F(x)')
        datos_1 = datos_1.iloc[:-1]

        Q_bi_prima = datos_1.append(datos_2.iloc[0]).sort_values('F(x)')
        #print(f"Generación:{conteo} : {Q_bi_prima['Genotipo'].iloc[0]} : {Q_bi_prima['F(x)'].iloc[0]}")
        generaciones.append('Generación : '+str(conteo))
        genotipos.append(Q_bi_prima['Genotipo'].iloc[0])
        result.append(Q_bi_prima['F(x)'].iloc[0])


        Pinic = list(Q_bi_prima['Genotipo'])
        conteo +=1
    datos_finales = pd.DataFrame({'Generaciones':generaciones, 'Genotipos':genotipos, 'F(x)':result})
    return datos_finales
```

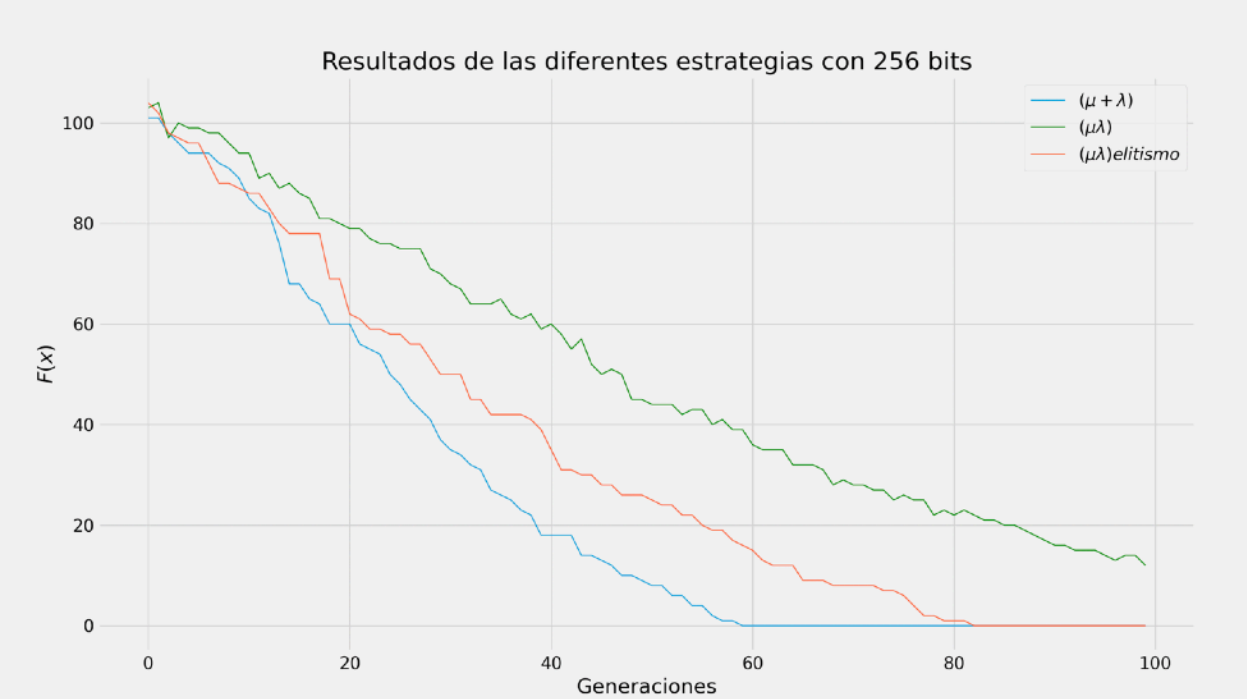
Resultados:

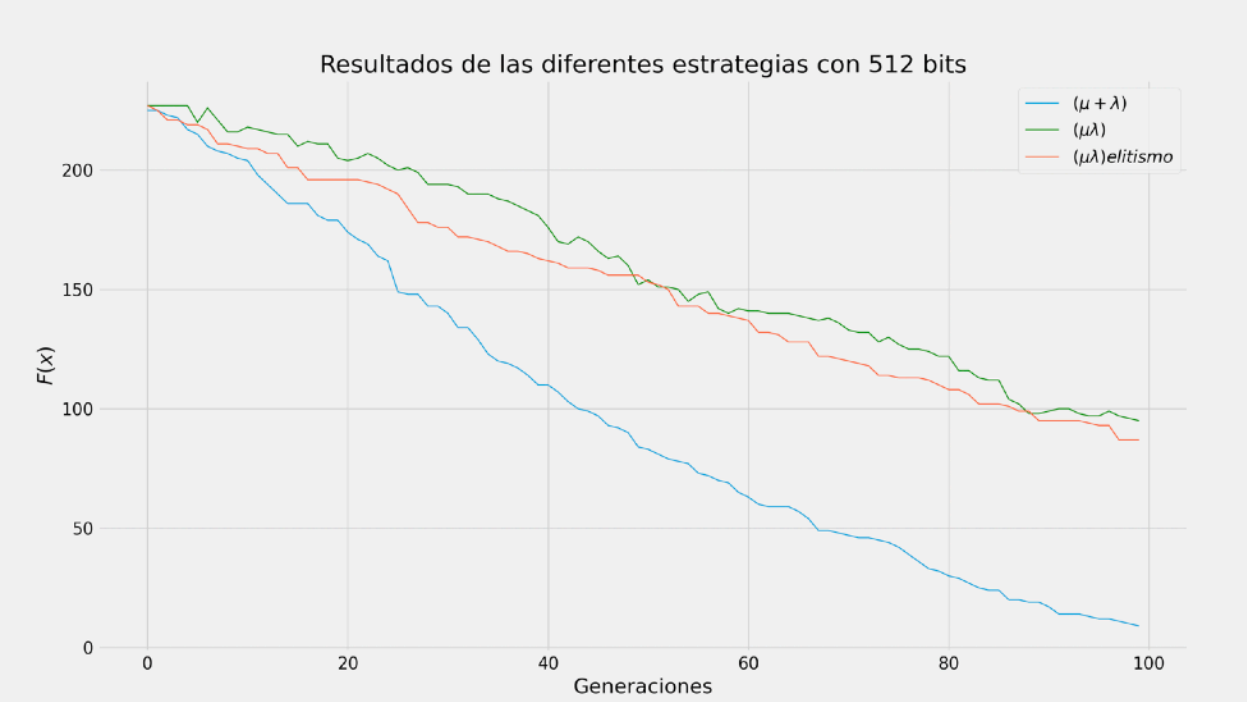
Dentro de las diferentes estrategias me di cuenta que si pudiéramos enumerar de mejor a pero estrategia según las generaciones que tarda en converger a 0 o lo más próximos posibles quedarían de la siguiente forma.

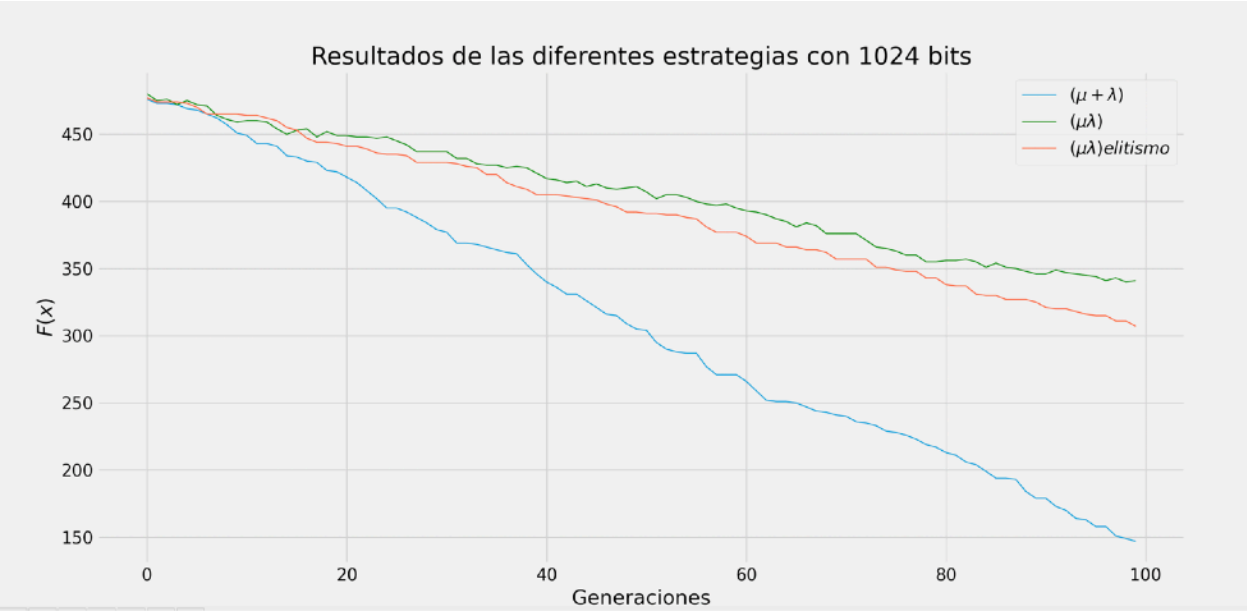
1. $(\mu + \lambda)$
2. $(\mu\lambda)\ con\ elitismo$
3. $(\mu\lambda)$

Debido a que la 1 los resultados son monótonos decrecientes, en la 2 aunque es decreciente sucede que durante varias generaciones tiene como resultado el mismo fenotipo por lo que parecería que se queda estancado algunas iteraciones y por último la 3 decrece con mucho ruido donde en algunas generaciones llega a suceder que el mejor fenotipo de la generación es peor que el mejor fenotipo de una iteración anterior.


Resultados de las diferentes estrategias con 128 bits
$(\mu + \lambda)$
$(\mu\lambda)$
$(\mu\lambda)elitismo$
$F(x)$
Generaciones
0
10
20
30
40
50
0
20
40
60
80
100


Resultados de las diferentes estrategias con 256 bits
$(\mu + \lambda)$
$(\mu\lambda)$
$(\mu\lambda)elitismo$
$F(x)$
Generaciones
0
20
40
60
80
100
0
20
40
60
80
100


Resultados de las diferentes estrategias con 512 bits
$(\mu + \lambda)$
$(\mu\lambda)$
$(\mu\lambda)elitismo$
$F(x)$
0
50
100
150
200
0
20
40
60
80
100
Generaciones


Resultados de las diferentes estrategias con 1024 bits
$(\mu + \lambda)$
$(\mu\lambda)$
$(\mu\lambda)elitismo$
$F(x)$
150
200
250
300
350
400
450
0
20
40
60
80
100
Generaciones

Apéndice:

Ejemplo de como imprime en pantalla el programa con 128 bits:

```
   Generaciones                                                  Genotipos  F(x)
 Generación : 0  [1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, ...   472
 Generación : 1  [1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, ...   471
 Generación : 2  [0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, ...   466
 Generación : 3  [1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, ...   464
 Generación : 4  [1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, ...   462
            ...                                                  ...   ...
Generación : 95  [1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, ...   186
Generación : 96  [0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, ...   185
Generación : 97  [1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, ...   176
Generación : 98  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, ...   170
Generación : 99  [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, ...   170
```