



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO



2024

SESIÓN DE LABORATORIO 5: ALGORITMOS GENÉTICOS PARA OPTIMIZACIÓN COMBINATORIA

ALGORITMOS BIOINSPIRADOS

Elaborado por: Flores Estopier Rodrigo

Boleta: 2021630260

Profesor: Rosas Trigueros Jorge Luis

Grupo: 6CV2

Fecha de realización: 18/10/2024

Fecha de entrega: 22/10/2024

Semestre: 25-1

Contenido

1. Marco Teórico 2

2. Desarrollo 2

 Codificación 3

 Ejecución 9

3. Conclusiones 12

4. Bibliografía 13

1. MARCO TEÓRICO

Un Algoritmo Genético (AG) es una técnica de optimización inspirada en el proceso de evolución biológica, donde una población de soluciones candidatas evoluciona iterativamente hacia una mejor solución. Los AG se basan en los principios de la selección natural de Darwin y emplean operadores genéticos como cruce y mutación para explorar el espacio de soluciones. Se utilizan ampliamente en problemas de optimización compleja, como el Problema del Viajero (TSP), donde se busca encontrar la ruta más corta que pase por un conjunto de ciudades una sola vez.

En el TSP, el objetivo es minimizar la distancia total recorrida al visitar cada ciudad exactamente una vez y regresar a la ciudad inicial. Los AG permiten abordar este problema combinando las mejores rutas generadas en cada generación y aplicando mutaciones aleatorias para mantener la diversidad genética y evitar el estancamiento en óptimos locales.

Operadores Genéticos:

1. **Ordered Crossover (OX):** El cruce ordenado es un operador genético diseñado específicamente para problemas donde el orden de los genes es importante, como el TSP. Se selecciona un segmento del primer padre que se transfiere al hijo, y los genes restantes se completan del segundo padre respetando el orden relativo de aparición.
2. **Mutación por Intercambio (Exchange Mutation):** Esta mutación consiste en intercambiar aleatoriamente dos genes (ciudades) del cromosoma. La mutación introduce variabilidad en la población, evitando la convergencia prematura hacia soluciones subóptimas.

2. DESARROLLO

Se modificará el código proporcionado en clase para adaptar el algoritmo genético para encontrar la solución óptima del problema del viajero, considerando 10 ciudades. Donde la distancia entre cada ciudad se representa en la siguiente matriz

0 1 2 3 4 5 6 7 8 9 10

0	[2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]
1	[2. 2. 2. 1. 2. 2. 2. 2. 2. 2. 2.]
2	[2. 2. 2. 2. 1. 2. 2. 2. 2. 2. 2.]
3	[2. 2. 2. 2. 2. 1. 2. 2. 2. 2. 2.]
4	[2. 2. 2. 2. 2. 2. 1. 2. 2. 2. 2.]
5	[2. 2. 2. 2. 2. 2. 2. 1. 2. 2. 2.]
6	[2. 2. 2. 2. 2. 2. 2. 2. 1. 2. 2.]
7	[2. 2. 2. 2. 2. 2. 2. 2. 2. 1. 2.]
8	[2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 1.]
9	[2. 2. 1. 2. 2. 2. 2. 2. 2. 2. 2.]
10	[2. 2. 2. 2. 2. 2. 2. 2. 2. 2. 2.]

Donde no se consideran los índices 0.

Así, por ejemplo, teniendo una posible solución como la siguiente [2,4,6,1,3,5,7,10,8,9] la distancia total recorrida es la siguiente. El objetivo del algoritmo es encontrar la menor distancia en la que podemos recorrer las 10 ciudades.

CODIFICACIÓN

El código comienza creando una matriz de distancias entre 11 ciudades utilizando numpy. La matriz es inicialmente poblada con valores de distancia de 2, pero ciertas conexiones entre ciudades tienen una distancia más corta (valor de 1). Este conjunto de distancias define el problema que se resuelve.

```
import numpy
M=2*numpy.ones([11,11])
M[1][3]=1; M[3][5]=1; M[5][7]=1; M[7][9]=1;
M[9][2]=1; M[2][4]=1; M[4][6]=1; M[6][8]=1;
M[8][10]=1

#Imprime la matriz con indices
print("Matriz de distancias")
#Imprime los indices de arriba
print(" ",end=" ")
for i in range(11):
    print(i,end=" ")
print()
```

```
#Imprime los indices de la izquierda
for i in range(11):
    print(i,M[i])
```

Entre las librerías que utilizaremos, se encuentran:

- Random: para la ruleta de elección y la generación de la población inicial.
- Numpy: Para la matriz de distancias entre ciudades.
- Functools, cmp_to_key: para utilizar una funcion de comparación como criterio de ordenamiento.

La longitud de los cromosomas es igual al número de ciudades (10 en este caso, excluyendo la ciudad de índice 0). Se inicializa una población de 100 cromosomas, cada uno representando un recorrido aleatorio por las ciudades.

```
L_chromosome= len(M[0])-1

#Numero de cromosoms
N_chromosomes=100
#probability of mutation
prob_m=0.05
```

Definimos una funcion para generar cromosomas de forma aleatoria.

```
def random_chromosome():
    global L_chromosome
    chromosome = list(range(1, L_chromosome + 1))
    random.shuffle(chromosome)
    return chromosome
```

Creamos nuestra población inicial utilizando la funcion de generación aleatoria de cromosomas, mientras que establecemos un valor de aptitud de 0 a todos los cromosomas.

```
#Inicializar poblacion inicial
for i in range(0,N_chromosomes):
    F0.append(random_chromosome())
    fitness_values.append(0)
```

Creemos la funcion de aptitud, que en este caso se calcula la distancia total recorrida por el recorrido representado por el cromosoma.

```
#Funcion de aptitud
def fun_apitud(chromosome):
    distance = 0
    for i in range(len(chromosome) - 1):
        distance += M[chromosome[i]][chromosome[i + 1]]
    distance += M[chromosome[-1]][chromosome[0]] # Regreso a la ciudad
    inicial
    return distance
```

La funcion para evaluar cromosomas utiliza la funcion la funcion de aptitud para obtener los valores de distancia total de cada cromosoma en la población actual para ingresarlos a una lista que representa la aptitud de cada cromosoma.

```
def evaluate_chromosomes():
    global F0
    for p in range(N_chromosomes):
        fitness_values[p]=fun_apitud(F0[p])
```

También, definimos la funcion de comparación de cromosomas, la cual recibe dos cromosomas y compara sus valores de aptitud (distancia). Esta funcion será utilizada como criterio para ordenar a los cromosomas por aptitud.

```
def compare_chromosomes(chromosome1,chromosome2):
    fvc1=fun_apitud(chromosome1)
    fvc2=fun_apitud(chromosome2)
    if fvc1 > fvc2:
        return 1
    elif fvc1 == fvc2:
        return 0
    else: #fvg1<fvg2
        return -1
```

Para elegir a los cromosomas para la generación de nuevas generaciones, implementamos un mecanismo de selección basado en la aptitud mediante una ruleta que da más probabilidades de ser seleccionados a los cromosomas con mejor aptitud. Primero definimos las variables:

- **Lwheel**: Define el tamaño de la "ruleta", donde cada cromosoma tendrá una fracción proporcional a su aptitud. El tamaño de la ruleta se multiplica por 10 para aumentar la granularidad de la selección.
- **maxv**: Valor máximo de aptitud en la población, que se utilizará para calcular las probabilidades relativas de los cromosomas.
- **acc**: Variable para acumular el "fitness relativo" de cada cromosoma en comparación con el valor máximo.

Posteriormente se calcula la fracción de la ruleta que corresponde a cada cromosoma. La fracción para cada cromosoma se obtiene dividiendo su aptitud relativa por el total acumulado (acc).

Si la fracción es demasiado pequeña, es ajustada a un mínimo de $1.0 / \text{Lwheel}$ para garantizar que ningún cromosoma tenga una probabilidad nula de ser seleccionado.

Después, se ajustan las fracciones de los dos primeros cromosomas para asegurarse de que la suma total de las fracciones sea exactamente 1.

Por último, construimos la ruleta, donde para cada fracción **f** calculada, se determina cuántos "espacios" (o "tickets") tiene ese cromosoma en la ruleta mediante **Np = int(f * Lwheel)**. Cromosomas con mayor aptitud ocupan más espacios en la ruleta, y, por lo tanto, tienen mayor probabilidad de ser seleccionados.

Luego, se añade el índice del cromosoma correspondiente (almacenado en **pc**) a la ruleta tantas veces como indique **Np**.

```
def create_wheel():
    global F0, fitness_values
    Lwheel=N_chromosomes*10
    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
```

Posteriormente, inicializamos la siguiente generación haciendo una copia de la generación actual. Y establecemos una variable n que nos permitirá saber cuantas generaciones se han creado.

```
F1=F0[:]
n=0
```

La función más importante de este algoritmo es la función de generar una siguiente generación, aquí es donde utilizamos operadores genéticos para crear más candidatos a una solución.

Primero, ordenamos los cromosomas de acuerdo con su aptitud utilizando como criterio la función de comparación de cromosomas. Además, la mejor solución es mostrada, imprimiendo la mejor ruta hasta el momento y el valor de la aptitud (la distancia recorrida).

```
def nextgeneration(b):
    global n
    #display.clear_output(wait=True)
    #display.display(button)
    F0.sort(key=cmp_to_key(compare_chromosomes) )
    #print( "Best solution so far:")
    n+=1
    print( n, " ", F0[0], "= ", fun_apitud(F0[0]) )
```


Posteriormente, reservamos los dos mejores cromosomas de la generación actual, para mantenerlos en la siguiente generación, además creamos la ruleta para seleccionar a los padres para la siguiente generación.

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

roulette=create_wheel()
```

Ahora realizamos el proceso de selección de padres, cruce ordenado (Ordered crossover) y mutación (Exchange mutation) para generar nuevos descendientes en la siguiente generación.

En un bucle donde el número de iteraciones es la mitad del tamaño de la población restante, porque en cada iteración se generan dos descendientes.

Con `random.choice(roulette)` seleccionamos aleatoriamente uno de los padres basándose en la ruleta, donde los cromosomas con mejor aptitud tienen más probabilidades de ser seleccionados.

El crossover ordenado se lleva a cabo en la siguiente función, este método asegura que el orden relativo de los genes se mantenga:

```
# Ordered crossover (OX)
def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child = [None] * len(parent1)
    child[start:end] = parent1[start:end]

    pointer = end
    for gene in parent2:
        if gene not in child:
            if pointer >= len(parent2):
                pointer = 0
            child[pointer] = gene
            pointer += 1
    return child
```

Después del cruce, se aplica mutación a cada hijo creado con una probabilidad `prob_m`.

Si la condición de mutación se cumple, se ejecuta la mutación tomando dos elementos aleatorio del cromosoma e intercambian su posición entre ellos.

```
# Exchange mutation
def mutate(chromosome):
    a, b = random.sample(range(len(chromosome)), 2)
    chromosome[a], chromosome[b] = chromosome[b], chromosome[a]
```

Finalmente, asignamos los descendientes a la siguiente generación. Y después generar todos los hijos remplazamos la generación actual con la nueva generación.

```
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
    o1=crossover(F0[p1],F0[p2])
    o2=crossover(F0[p2],F0[p1])
    #Each descendant is mutated with probability prob_m
    #print(o1)
    if random.random() < prob_m:
        mutate(o1)


    #print("Mutacion? ",o1)
    if random.random() < prob_m:
        mutate(o2)
    #The descendants are added to F1
    F1[2+2*i]=o1
    F1[3+2*i]=o2

#The generation replaces the old one
F0[:]=F1[:]
```

EJECUCIÓN

Realizamos pruebas considerando los siguientes parámetros para el algoritmo genético:

- Población: 100
- Probabilidad de mutación : 0.05
- No de generaciones: 500



Matriz de distancias

	0	1	2	3	4	5	6	7	8	9	10
0	[2.	2.	2.	2.	2.	2.	2.	2.	2.	2.	2.]
1	[2.	2.	2.	1.	2.	2.	2.	2.	2.	2.	2.]
2	[2.	2.	2.	2.	1.	2.	2.	2.	2.	2.	2.]
3	[2.	2.	2.	2.	2.	1.	2.	2.	2.	2.	2.]
4	[2.	2.	2.	2.	2.	2.	1.	2.	2.	2.	2.]
5	[2.	2.	2.	2.	2.	2.	2.	1.	2.	2.	2.]
6	[2.	2.	2.	2.	2.	2.	2.	2.	1.	2.	2.]
7	[2.	2.	2.	2.	2.	2.	2.	2.	2.	1.	2.]
8	[2.	2.	2.	2.	2.	2.	2.	2.	2.	2.	1.]
9	[2.	2.	1.	2.	2.	2.	2.	2.	2.	2.	2.]
10	[2.	2.	2.	2.	2.	2.	2.	2.	2.	2.	2.]

Ilustración 1. Visualización de la matriz de distancias.

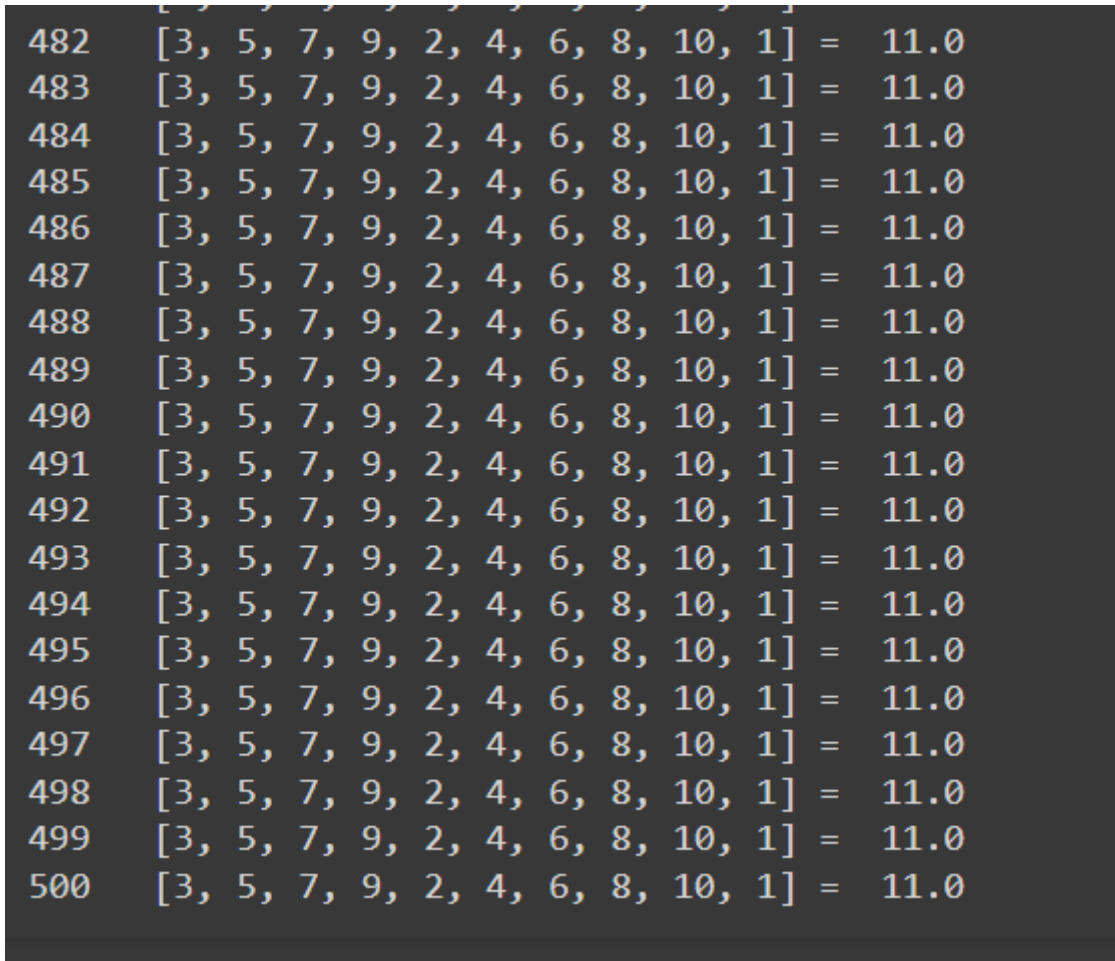
```
#Realizar distintas iteraciones
```

```
for i in range(500):
```

```
    nextgeneration(0)
```

```
1  [7, 9, 6, 8, 2, 1, 4, 10, 3, 5] = 16.0
2  [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
3  [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
4  [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
5  [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
6  [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
7  [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
8  [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
9  [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
10 [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
11 [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
12 [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
13 [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
14 [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
15 [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
16 [3, 6, 8, 10, 2, 4, 9, 5, 7, 1] = 15.0
17 [10, 1, 3, 5, 2, 7, 9, 4, 6, 8] = 14.0
18 [10, 1, 3, 5, 2, 7, 9, 4, 6, 8] = 14.0
19 [10, 1, 3, 5, 2, 7, 9, 4, 6, 8] = 14.0
20 [10, 1, 3, 5, 2, 7, 9, 4, 6, 8] = 14.0
```

Ilustración 2. Primera sección de la ejecución del algoritmo.



482	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
483	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
484	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
485	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
486	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
487	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
488	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
489	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
490	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
491	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
492	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
493	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
494	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
495	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
496	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
497	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
498	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
499	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0
500	[3, 5, 7, 9, 2, 4, 6, 8, 10, 1]	=	11.0

Ilustración 3. Última sección del algoritmo genético, donde se encuentra una solución óptima.

3. CONCLUSIONES

En esta práctica, se utilizó un algoritmo genético para resolver el problema del viajero, optimizando la ruta entre 10 ciudades. El uso de operadores genéticos como el crossover ordenado y la mutación por intercambio permitió explorar diferentes rutas y minimizar la distancia total. Una de las observaciones más importantes fue que, aunque las soluciones iniciales eran aleatorias y subóptimas, el algoritmo fue capaz de mejorar las rutas de manera constante a lo largo de las generaciones.

Una sugerencia para mejorar el desarrollo de la práctica sería experimentar con diferentes tasas de mutación y tamaños de población para estudiar su impacto en la convergencia de la solución. Adicionalmente, la inclusión de métodos de visualización

más detallados de las rutas y sus distancias permitiría comprender mejor el progreso del algoritmo.

Un experimento realizado fue incrementar el número de generaciones y ajustar la probabilidad de mutación. Se observó que una tasa de mutación muy alta afectaba la estabilidad de las soluciones, mientras que una tasa moderada (5%) mantenía un equilibrio entre exploración y explotación del espacio de soluciones. A través de este aprendizaje, se concluye que el ajuste adecuado de los parámetros del AG es crucial para lograr una optimización eficiente.

4. BIBLIOGRAFÍA

- [1] “Genetic Operations”, Uab.cat. [En línea]. Disponible en:
<https://mat.uab.cat/~alseda/MasterOpt/GeneticOperations.pdf>. [Consultado: 19-oct-2024].
- [2] chitrankmishra Follow Improve, “Traveling Salesman Problem using Genetic Algorithm”, GeeksforGeeks, 07-feb-2020. [En línea]. Disponible en:
<https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/>. [Consultado: 19-oct-2024].