Instituto Politécnico Nacional

Escuela Superior de Cómputo

Evolutionary Computing

Lab Session 4: GA's for Combinatorial Optimization

Alumno: David Arturo Oaxaca Pérez

Grupo: 3CV11

Profesor: Jorge Luis Rosas Trigueros

Fecha de realización: 09/10/2021

Fecha de entrega: 14/10/2021

Marco teórico

En esta práctica se emplearán nuevamente algoritmos genéticos para la optimización, pero en este caso, estarán enfocados hacia la combinación de elementos en un conjunto para conseguir el mejor resultado, que dependerá según el problema.

Estos algoritmos son una serie de pasos que describe un proceso a seguir para llegar a una solución óptima, inspirados en la evolución natural y la evolución genética, empiezan con un conjunto inicial de soluciones potenciales codificadas que suelen ser generadas aleatoriamente, a las que usualmente les llaman cromosomas. Estos se depurarán hasta escoger la mejor solución mediante mecanismos de presión selectiva. Cada una de estas soluciones potenciales es evaluada por una función de aptitud para determinar si es más apta con respecto a las demás soluciones, esto se hará por varias iteraciones o generaciones hasta que se llegue a un criterio para detener el algoritmo, que en muchos casos será el número de generaciones iteradas.

En este caso tuvimos que plantear una solución propia para un problema, con base a algunos operadores genéticos vistos en clase, de igual manera para lograrlo empleamos los 4 pasos vistos que son fundamentales para definir un algoritmo genético:

- Codificar una solución potencial en un cromosoma
- Diseñar una función fitness para determinar la aptitud de los cromosomas
- Aplicar operadores genéticos (En este caso, mutación y crossover)
- Presión selectiva

Problema de la mochila (Knapsack problem) 0-1

Este problema es el que habíamos trabajado previamente usando programación dinámica en otra sesión de laboratorio, solo que en esta usaremos un algoritmo genético para llegar a la solución más óptima.

En este problema tenemos una “mochila” con un peso W y una lista de elementos donde cada uno tiene un peso w y un valor v, para llegar a la solución óptima tenemos que escoger el conjunto de elementos que nos aporte el mayor valor posible sin sobrepasar el peso de la mochila. Su función está dada por:

Maximizar

$$\sum_{i=1}^{n} v_i x_i$$

Sujeto a

$$\sum_{i=1}^{n} w_i x_i \leq W \; and \; x_i \; \in \{0,1\}$$

Para este problema también se siguen los pasos establecidos:

Codificar una solución potencial en un cromosoma

El cromosoma será una cadena con la misma cantidad de bits que los ítems que vamos a comparar para ver cuál entra en la mochila. Cada bit será 0 o 1, 1 si metemos el item a la mochila, 0 si no lo metemos.

Es decir, si tenemos los siguientes datos:

$$w = (2, 4, 5, 5, 7)$$

$$v = (2, 3, 4, 5, 6)$$

$$W=9$$

Un cromosoma será como el siguiente:

$$(0\ 1\ 1\ 0\ 1)$$

Diseñar una función fitness para determinar la aptitud de los cromosomas

En este caso la función fitness será parecida a la función matemática que define el problema declarada previamente, sumaremos los valores de los ítems que entran en la mochila según el cromosoma y también sus pesos, si el peso excede el de la mochila le restaremos al valor total el exceso y una penalización arbitraria para indicar que no va por el camino correcto. Entre mayor sea el valor, más apto es el cromosoma.

Aplicar operadores genéticos (En este caso, mutación y crossover)

En este caso se aplicarán dos operadores:

Mutación: Con base a una probabilidad previamente definida, se escogerá un bit al azar del cromosoma para ser cambiado y agregar diversidad al nuevo cromosoma. Ejemplo:

$$(0\ 1\ 1\ 1\ 0) \rightarrow (0\ 1\ 1\ 0\ 0)$$

Crossover: En este operador los padres heredan la mitad de su cadena a su hijo (offspring) que formara parte de la nueva generación. Ejemplo:

El punto de crossover será el largo del cromosoma entre dos.

$$(0\ 1\ 1 \mid 0\ 1), (1\ 0\ 0 \mid 1\ 1) \rightarrow (0\ 1\ 1 \mid 1\ 1), (1\ 0\ 0 \mid 0\ 1)$$

Presión selectiva

Para este paso usaremos el elitismo, haciendo que los dos cromosomas más aptos pasen a la nueva generación y el resto se recombinen para que la población esté compuesta en su mayoría por nuevos cromosomas. Además de esto, entre más

apto se el cromosoma, más grande será su fracción en la rueda haciendo que sea más probable que parte de su composición pase a las nuevas generaciones.

Problema del viajero (Travelling Salesman Problem)

Este problema consiste en la siguiente pregunta: “Dada una lista de ciudades y las distancias entre cada par de ciudades, ¿Cuál es la ruta más corta posible de manera que se visiten todas las ciudades exactamente una vez y se regrese a la ciudad de origen?”. Existen muchas variaciones del problema, por ejemplo, cuando se tiene una ciudad origen predefinida o cuando las distancias de ida y vuelta entre una ciudad y otra son diferentes, pero en el problema resuelto en esta sesión de laboratorio cualquier ciudad puede ser la de origen y la ida y vuelta a una ciudad es de la misma distancia.

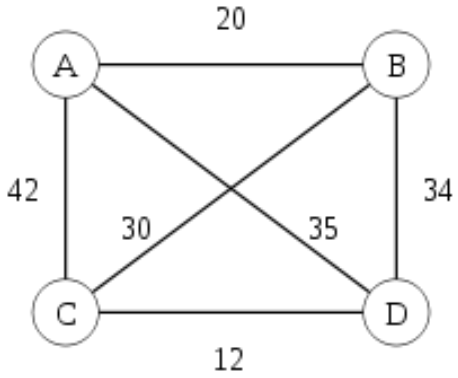


Figura 1. Grafo representando las ciudades y sus distancias en el problema del viajero.

Para este problema también se siguen los pasos previamente establecidos:

Codificar una solución potencial en un cromosoma

El cromosoma será un recorrido por todas las ciudades, dando a entender que, si el número de ciudades es 7, el cromosoma estará compuesto por 7 elementos en el que cada uno representa una ciudad, además de que no se podrá repetir ninguno de estos elementos.

Un cromosoma será como el siguiente:

(1 2 3 4 5 6 7)

Siendo este un recorrido por las 7 ciudades.

Diseñar una función fitness para determinar la aptitud de los cromosomas

Para este problema primero tenemos que considerar cual será la decodificación del problema. Lo que se hará en esta, será crear una lista con las distancias entre cada

par de ciudades y la última ciudad con la primera para completar el recorrido, posteriormente lo que hará la función fitness será retornar la suma de estas distancias y entre menor sea el valor, será más apto, pues el problema consiste en encontrar el recorrido completo de menor distancia.

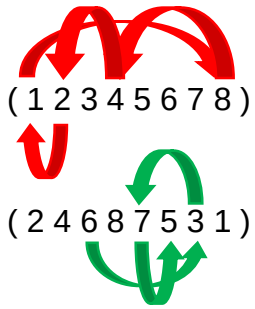
Aplicar operadores genéticos (En este caso, mutación y crossover)

Aplicaremos dos operadores:

Mutación: En este caso se usará una “Exchange Mutation (EM)”, o mutación de intercambio, con base a una probabilidad previamente definida, se intercambiarán dos ciudades de lugar para darle a los cromosomas mayor variedad. Ejemplo:

(1 2 3 4 5 6 7) -> (1 2 5 4 3 6 7)

Crossover: Para este operador se usará el “Cycle Crossover (CX)” o crossover de ciclo, en este partiremos con el primer elemento de uno de los padres, pasaremos al último e iremos comparando si el elemento no se encuentra ya en el hijo para ir añadiéndolo, posterior a eso, nos moveremos a la posición del elemento añadido en el padre del que no se escogió dicho elemento, un ejemplo más claro de este tipo de crossover es el siguiente:



Offspring:

(1 2 6 4 7 5 3 8)

De esa manera vamos recorriendo ambos cromosomas y creando una combinación en el hijo, claro que hay un par de casos especiales, donde se añaden los valores que en el siguiente salto encontraremos en los cromosomas de manera inversa y al no hacerse manualmente es difícil saber si se debió tomar el otro valor, así que se propuso como modificación añadir uno de los valores no usados si se llegaba a las condiciones que creaban un bucle sin fin por alguna elección como escoger la ciudad del segundo padre en vez de la del primero en el último bit para el siguiente caso:

(3 7 5 4 6 2 1) y (2 3 5 4 6 1 7)

Presión selectiva

En esta usaremos el elitismo, haciendo que los dos cromosomas más aptos pasen a la nueva generación y el resto se recombinen para que la población esté compuesta en su mayoría por nuevos cromosomas. De igual manera se usará la rueda con una mayor fracción para los cromosomas más aptos.

Las ciudades y distancias para este problema usadas fueron:

	Berlín	Cairo	Chicago	Honolulu	London	CDMX	Montreal
Berlín	-	1,795	4,405	7,309	579	6,047	3,729
Cairo	1,795	-	6,129	8,838	2,181	7,688	5,414
Chicago	4,405	6,129	-	4,250	3,950	1,692	744
Honolulu	7,309	8,838	4,250	-	7,228	3,779	4,910
London	579	2,181	3,950	7,228	-	5,550	3,282
CDMX	6,047	7,688	1,691	3,779	5,550	-	2,318
Montreal	3,729	5,414	744	4,910	3,282	2,318	-

Tabla 1. Ciudades y distancias usadas para el problema del viajero (TSP).

Material y equipo

- El programa fue realizado en Google Colaboratory usando Python 3.9 como lenguaje.
- El sistema que se utilizó para acceder a Google Colaboratory fue una laptop de marca Lenovo, modelo Ideapad S540 con Ryzen 7 y 8 GB de RAM, con sistema operativo Windows 10.

Desarrollo de la practica

Problema de la mochila (Knapsack problem) 0-1

Para el problema de la mochila utilizamos el código que fue mostrado en clase para la sesión de laboratorio, pero se realizando algunas modificaciones puestas como requisitos, entre ellos los valores para los que se ejecutaría el programa.

El primer cambio se hizo en la creación de los valores y pesos de los ítems en la mochila, en este caso necesitábamos tener una mochila con un peso W = 1 y 20 objetos cuyos pesos fueran creados de manera aleatoria en decimales entre el 0 y el 1, es decir, (0, 1) y sus valores, que también fueran aleatoria, estuvieran entre el 1 y el 100 de manera inclusiva [0, 100].

Para hacer esto usamos propiedades del import random de Python como randint, que trae elementos desde el primer parámetro hasta el anterior del segundo parámetro que funciona como un límite (Funciona como [a, b-1]) y de random() que trae un numero aleatorio entre el 0 y el 1.

```
#Random creation of 20 values an weights for the items
v = [random.randint(1, 100) for _ in range(20)]
w = [round(random.random(), 3) for _ in range(20)]
W = 1
```

Figura 2. Variables usadas para el problema de la mochila, creando los pesos y valores de los ítems de manera aleatoria.

Otro cambio importante para poder llegar a una solución correcta fue el uso de penalizaciones, como no había una previamente era posible que un cromosoma que excedía el peso de la mochila fuera el más apto, ya que al no respetar esa limitante podía “tomar” una mayor cantidad de ítems para aumentar su valor, lo cual nos llevaba a soluciones incorrectas muchas veces.

Primero planteamos un valor de penalización algo grande para asegurarnos de que efectivamente no se consideren aptos los cromosomas que propongan una solución que no cumple con los requisitos del problema.

```
#Penalty value to punish solutions that surpass the maximum weight
penalty_value = 100000
```

Figura 3. Valor de penalización usado para el problema de la mochila.

Posteriormente aplicamos esta penalización en la función fitness a todo cromosoma cuyo peso exceda el de la mochila.

```
def fitness_function(x):
  global W
  Total_value,Total_weight=x
  excess = Total_weight-W
  return Total_value if excess <= 0 else (Total_value - (excess + penalty_value))
```

Figura 4. Implementación de una penalización en la función fitness si el peso de una solución excede el de W.

A continuación, podemos ver los resultados obtenidos:

```
Poblacion: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
Valores fitness: [-98993.936, -98998.227, -99269.438, -99546.21, -99259.483, -99240.956, -99483.317, -99192.996]

Best solution so far:
Generacion 288, Cromosoma mas apto: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
f( (1015, 9.936) )= -98993.936

Poblacion: [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
Valores fitness: [-98993.936, -98998.227, -99240.956, -99300.805, -99511.799, -99164.514, -99258.335, -99474.793]

Best solution so far:
Generacion 289, Cromosoma mas apto: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
f( (1015, 9.936) )= -98993.936
```

Figura 5. Primeros resultados con un valor penalizado para el problema de la mochila

```
Poblacion: [[0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
Valores fitness: [292, -98993.936, -98993.936, -99218.955, -99183.371, -99055.964, -99325.107, -99065.285]

Best solution so far:
Generacion 998, Cromosoma mas apto: [0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
f( (292, 0.9190000000000002) )= 292

Poblacion: [[0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
Valores fitness: [292, -98993.936, -99193.909, -99125.134, -98993.936, -98993.936, -99065.285, -99193.909]

Best solution so far:
Generacion 999, Cromosoma mas apto: [0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
f( (292, 0.9190000000000002) )= 292

Poblacion: [[0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
Valores fitness: [292, -98993.936, -99287.73, -99055.964, -99038.885, -99193.909, -99387.135, -99065.914]

Best solution so far:
Generacion 1000, Cromosoma mas apto: [0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
f( (292, 0.9190000000000002) )= 292
```

Figura 6. Resultados obtenidos después de 1000 generaciones para el problema de la mochila.

Parte de las dificultades fue establecer una penalización que efectivamente hiciera que las soluciones que excedieran el peso de la mochila fueran descartadas, pues restar el exceso al valor no era suficiente ya que muchas veces los valores eran más grandes que el exceso de peso.

Algunas pruebas que se hicieron para ver si el algoritmo efectivamente estaba llegando al resultado óptimo fue tomar algunos de los valores aleatorio creados, multiplicar por 1000 todo lo respectivo al peso (Si W = 1 entonces W = 1000 y si wi = 0.500 entonces wi = 500) y resolverlo usando programación dinámica.

Para los siguientes datos:

```
v = [73, 22, 26, 91, 99, 52, 100, 61, 87, 64, 73, 58, 59, 11, 54, 98, 85, 39, 15, 88]
w = [832, 913, 693, 156, 247, 51, 928, 885, 890, 126, 232, 864, 495, 830, 325, 613, 775, 501, 766, 60]
W = 1000
N = len(v)
```

Figura 7. Datos empleados para la resolución especifica de un caso en el problema de la mochila.

Usando programación dinámica llegamos al siguiente resultado:

```
Maximo valor en la mochila: 467.0
Lista de los indices de los elementos tomados: [60, 232, 126, 51, 247, 156]
```

Figura 8. Resultado de resolver el problema mediante programación dinámica.

Probando el algoritmo genético no llegamos inicialmente a esta solución y tampoco nos acercamos tanto a esta, hasta que hicimos algunas modificaciones. Usando 40

cromosomas y 10,000 generaciones lo cual nos da un costo un poco más alto, pero efectivamente llegamos a la solución más óptima o una bastante cercana. Además de cargar un poco las probabilidades al crear un nuevo cromosoma hacia 0 (que no tome el objeto) y aumentar la probabilidad de mutación un 0.8 para asegurarnos de que haya mayor diversidad en las nuevas generaciones.

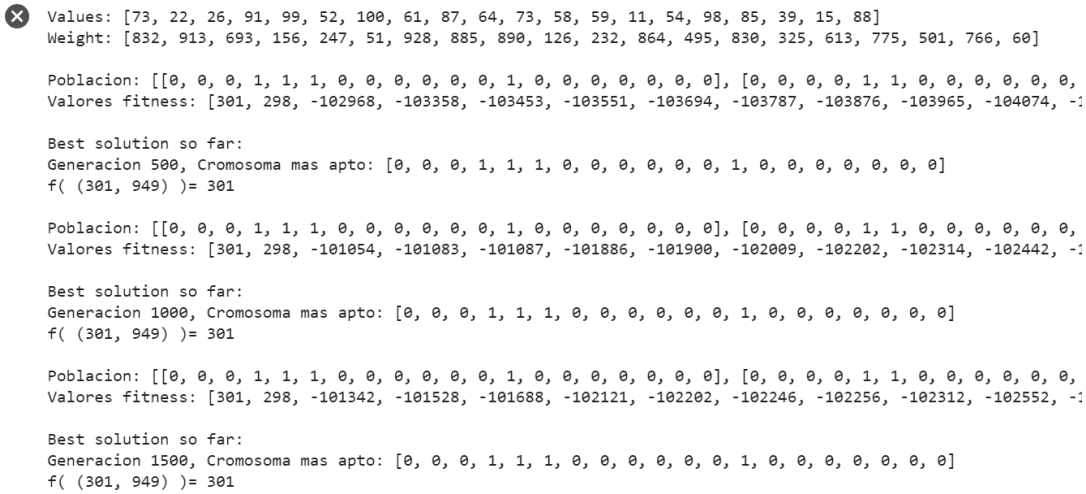


Figura 9. Primeras generaciones y datos usando el algoritmo genético.

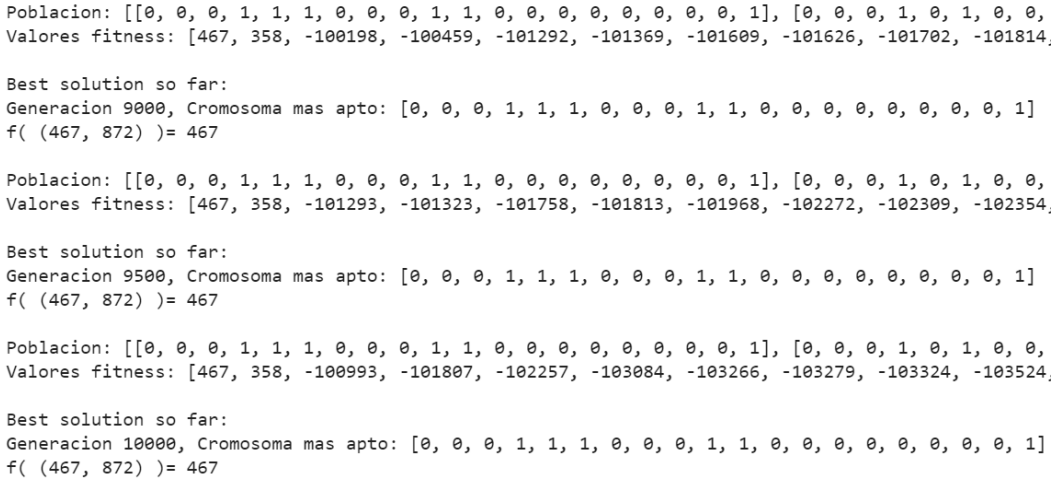


Figura 10. Últimas generaciones con el resultado más óptimo.

Probando nuevamente con valores aleatorios tenemos lo siguiente:

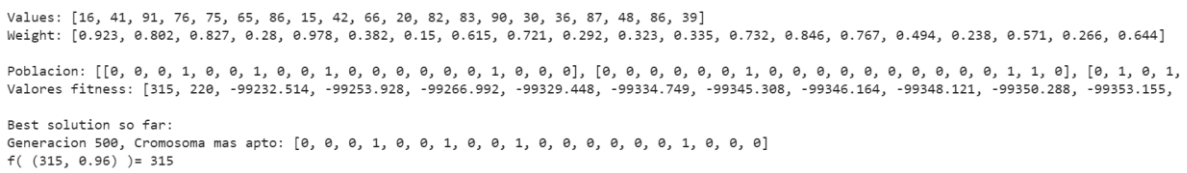


Figura 11. Nueva prueba con el algoritmo genético.

```
Poblacion: [[0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0], [0, 0, 0, 1, 0, 0, 1, 0, (
Valores fitness: [335, 315, -99212.464, -99263.291, -99264.582, -99319.462, -99324.511, -99344.969,

Best solution so far:
Generacion 9500, Cromosoma mas apto: [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]
f( (335, 0.934) )= 335

Poblacion: [[0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0], [0, 0, 0, 1, 0, 0, 1, 0, (
Valores fitness: [335, 315, -99233.137, -99236.834, -99269.275, -99290.979, -99303.316, -99357.443,

Best solution so far:
Generacion 10000, Cromosoma mas apto: [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0]
f( (335, 0.934) )= 335
```

Figura 12. Últimas generaciones del algoritmo genético con la nueva prueba.

Y multiplicando los valores aleatorios para resolverlo por programación dinámica obtenemos lo siguiente:

```
Maximo valor en la mochila: 341.0
Lista de los indices de los elementos tomados: [266, 238, 335, 150]
```

Figura 12. Resultado de la nueva prueba usando programación dinámica

Podemos ver que los resultados obtenidos por el algoritmo genético efectivamente ya son mucho más cercanos a la solución óptima y si bien tarda más en resolverse, está muy lejos de tener una complejidad temporal tan alta como la de fuerza bruta ($O(2^n)$) y también tiene una complejidad espacial menor a la solución de programación dinámica, pues solamente almacenamos los cromosomas y los valores de la función fitness pero no una tabla, entre otros beneficios, como el poder resolver el problema para pesos que pueden estar en decimales, cosa que por medio del enfoque “Bottom Up” no se puede hacer como tal sin hacer algunos cambios.

Tal vez una opción para llegar a esta solución más óptima y usar menos generaciones y menos cromosomas seria usar otros operadores genéticos, más métodos de presión selectiva o incluso otra clase de crossover o mutación, pero por el momento, estas fueron las pruebas realizadas.

Problema del viajero (Travelling Salesman Problem)

Para este problema también usamos como base el código visto en clase, pero en el si se tuvieron que hacer varias modificaciones para ajustarlo al problema que estábamos resolviendo, justo como están descritas en el marco teórico.

Lo primero que hicimos fue crear una tabla con las distancias de las ciudades que usamos para decodificar los cromosomas, esta es la que se encuentra en el marco teórico pero construida en un arreglo bidimensional.

```
#Tabla de ciudades y distancias entre ellas
cities_distances = [ [None, 1795, 4405, 7309, 579, 6047, 3729],
                     [1795, None, 6129, 8838, 2181, 7688, 5414],
                     [4405, 6129, None, 4250, 3950, 1691, 744],
                     [7309, 8838, 4250, None, 7228, 3779, 4910],
                     [579, 2181, 3950, 7228, None, 5550, 3282],
                     [6047, 7688, 1691, 3779, 5550, None, 2318],
                     [3729, 5414, 744, 4910, 3282, 2318, None]]
```

Figura 13. Tabla de las distancias entre ciudades para el TSP.

Posteriormente definimos los valores de las variables necesarias para el algoritmo, por ejemplo, tendremos que el tamaño del cromosoma será el número de ciudades, esto lo sacamos de la tabla de distancias ya que tanto el número de filas como de columnas será la cantidad de ciudades que hay, a esto le añadimos el número de cromosomas que usaremos y la probabilidad de mutación, etc.

```
#Chromosomes are 4 bits long
L_chromosome= len(cities_distances)
N_chains=2**L_chromosome
#Number of chromosomes
N_chromosomes=7
#probability of mutation
prob_m=0.75
#Población inicial
F0=[]
#Arreglo de valores de aptitud
fitness_values=[]
#Puntos en los que se hara el crossover
#(En este caso en crossover se hara por ciclos asi que no usaremos estos)
#crossover_start=2
#crossover_end=4
#Suma de todos los cromosomas
suma=float(N_chromosomes*(N_chromosomes+1))/2.
#Particiones de la rueda
Lwheel=N_chromosomes*10
```

Figura 14. Valores de las variables para el algoritmo genético.

La forma en que se creará un cromosoma será mediante un recorrido aleatorio por las ciudades, es decir, números del 1 al 7 que no se pueden repetir para representar el paso por cada ciudad.

```
def random_chromosome():
    chromosome=random.sample(range(1,8),7)
    return chromosome
```

Figura 15. Función para la creación de un cromosoma con un recorrido aleatorio.

En cuanto al manejo de la función fitness, esta se hará como se planteó en el marco teórico, primero obtendremos las distancias entre cada ciudad y en la función fitness retornaremos esta suma de distancias, que entre menor sea será considerada más apta.

```python
def decode_chromosome(chromosome):
  global L_chromosome

  # Distancias entre ciudades
  distances = []
  i,j = 0, 1
  while j < len(chromosome):
    #Con los contadores avanzamos en dos ciudades e iremos buscando
    #las distancias en la tabla
    city_1,city_2 = chromosome[i], chromosome[j]
    distances.append(cities_distances[city_1-1][city_2-1])
    i+=1
    j+=1

  #Añadimos la distancia de la ultima ciudad a la primera para que sea un
  #recorrido completo
  distances.append(cities_distances[chromosome[0]-1][chromosome[i]-1])

  return distances

def fitness_function(distances):
  #Retornamos la suma de todas las distancias
  return sum(distances)
```

Figura 16. Funciones para decodificar y aplicar la función fitness del algoritmo.

Otra parte importante a la hora de desarrollar este algoritmo fue el desarrollo de los operadores genéticos, la creación de la función de crossover fue la que se complicó un poco pues se hizo por medio de CX y a veces las decisiones que se iban tomando al agarrar una ciudad para el offspring llevaban a un ciclo sin fin, por ejemplo, se le daba preferencia a la ciudad del segundo cromosoma siempre y cuando esta no estuviera ya en el offspring, esto llevaba a que si en el siguiente paso las dos ciudades previamente escogidas eran las únicas opciones para escoger la siguiente ciudad a recorrer del offspring, no se podía avanzar.

Para solucionar esto se hizo una lista con el recorrido del cromosoma uno (Para nivelar que se le da preferencia al cromosoma 2 en la selección de ciudades) a la cual se le van quitando los elementos que se añaden al offspring y si llegamos a uno de esos ciclos en los que ya no se puede seleccionar se agarra la siguiente ciudad de esta lista para seguir avanzando.

En cuanto a la mutación, este fue un operador genético más fácil de codificar, simplemente tomamos dos índices al azar e intercambiamos las ciudades en estos.

```python
# Exchange Mutation (EM)
def mutation(Chromosome):

  change_1, change_2 = random.sample(range(0,len(cities_distances)),2)
  Chromosome[change_1], Chromosome[change_2] = Chromosome[change_2], Chromosome[change_1]

  return Chromosome
```

Figura 17. Función para generar una mutación en el cromosoma.

```
#Recorremos los cromosomas hasta que todos los elems del offspring tengan valor
while None in offspring:
  #Si llegamos a un index que ya recorrimos terminamos un ciclo
  while not (Chromosome2[leap] in offspring and Chromosome1[leap] in offspring):
    #Checamos si el elemento de uno de los cromosomas padres ya se encuentra
    if not (Chromosome2[leap] in offspring):
      #Si el elemento del segundo cromosoma no esta en el offspring, se selecciona
      offspring[leap] = Chromosome2[leap]
      free_elems.remove(offspring[leap])
      leap = Chromosome1.index(Chromosome2[leap])
    elif not (Chromosome1[leap] in offspring):
      #Si el elemento del primer cromosoma no esta en el offspring, se selecciona
      offspring[leap] = Chromosome1[leap]
      free_elems.remove(offspring[leap])
      leap = Chromosome2.index(Chromosome1[leap])
  else:
    # Buscamos el siguiente elemento del offspring que es nulo
    if Chromosome2[leap] in offspring and Chromosome1[leap] in offspring and offspring[leap] == None:
      #Si encontramos que ambas opciones para el siguiente nulo no son elegibles
      #tomamos la siguiente ciudad elegible del cromosoma 1
      offspring[leap] = free_elems[0]
      free_elems.remove(offspring[leap])
    elif None in offspring:
      leap = offspring.index(None)
```

Figura 18. Función para el crossover entre dos cromosomas.

Finalmente obtenemos los siguientes resultados al correr el algoritmo:

```
Best solution so far:
Poblacion: [[4, 5, 2, 1, 7, 3, 6], [1, 5, 6, 7, 4, 3, 2], [3, 6, 2, 7, 5, 1, 4], [3, 5, 2, 6, 7, 1, 4],
Valores fitness: [21147, 25531, 30213, 31425, 32835, 36531, 38377]

Best solution so far:
Generacion 1, Cromosoma mas apto: [4, 5, 2, 1, 7, 3, 6]
f(7228, 2181, 1795, 3729, 744, 1691, 3779)=  21147

Best solution so far:
Poblacion: [[4, 5, 2, 1, 7, 3, 6], [3, 5, 2, 1, 7, 4, 6], [1, 5, 6, 7, 4, 3, 2], [4, 6, 2, 1, 7, 3, 5],
Valores fitness: [21147, 22035, 25531, 28913, 29291, 36531, 36531]

Best solution so far:
Generacion 2, Cromosoma mas apto: [4, 5, 2, 1, 7, 3, 6]
f(7228, 2181, 1795, 3729, 744, 1691, 3779)=  21147

Best solution so far:
Poblacion: [[4, 5, 2, 1, 7, 3, 6], [3, 5, 2, 1, 7, 4, 6], [1, 5, 2, 4, 7, 3, 6], [3, 4, 2, 1, 7, 5, 6],
Valores fitness: [21147, 22035, 24990, 29135, 30592, 30878, 36531]

Best solution so far:
Generacion 3, Cromosoma mas apto: [4, 5, 2, 1, 7, 3, 6]
f(7228, 2181, 1795, 3729, 744, 1691, 3779)=  21147
```

Figura 19. Primeras generaciones para el TSP planteado.

```
Best solution so far:
Poblacion: [[4, 2, 1, 5, 7, 3, 6], [4, 2, 1, 5, 7, 3, 6], [4, 2, 1, 5, 7, 3, 6], [4, 2, 1, 5, 7, 6, 3],
Valores fitness: [20708, 20708, 20708, 22753, 29705, 30764, 36531]

Best solution so far:
Generacion 198, Cromosoma mas apto: [4, 2, 1, 5, 7, 3, 6]
f(8838, 1795, 579, 3282, 744, 1691, 3779)=  20708

Best solution so far:
Poblacion: [[4, 2, 1, 5, 7, 3, 6], [4, 2, 1, 5, 7, 3, 6], [4, 2, 1, 5, 7, 3, 6], [7, 2, 1, 5, 4, 3, 6],
Valores fitness: [20708, 20708, 20708, 23275, 29398, 29705, 36531]

Best solution so far:
Generacion 199, Cromosoma mas apto: [4, 2, 1, 5, 7, 3, 6]
f(8838, 1795, 579, 3282, 744, 1691, 3779)=  20708

Best solution so far:
Poblacion: [[4, 2, 1, 5, 7, 3, 6], [4, 2, 1, 5, 7, 3, 6], [4, 2, 1, 5, 7, 3, 6], [4, 2, 1, 5, 7, 3, 6],
Valores fitness: [20708, 20708, 20708, 20708, 22165, 22998, 36531]

Best solution so far:
Generacion 200, Cromosoma mas apto: [4, 2, 1, 5, 7, 3, 6]
f(8838, 1795, 579, 3282, 744, 1691, 3779)=  20708
```

Figura 20. El resultado más óptimo después de 200 generaciones para el TSP.

Hay que considerar que de la última ciudad se vuelve a la primera, es decir, el recorrido del cromosoma (4, 2, 1, 5, 7, 3, 6) es 4-2-1-5-7-3-6-4.

Algunos elementos interesantes que pude notar haciendo pruebas es que en la solución óptima siempre suelen estar los pares que tienen menor distancia, es decir 7 a 3 y 1 a 5, ya que ambas tienen distancias menores a 1000 millas de distancia, además de que se pueden obtener varias soluciones que llevan al mismo resultado, pero en todas ellas tenemos estos pares dentro. Otro asunto es que, el resultado no mejora en numerosas generaciones posteriores, sino que es lo bastante óptimo para permanecer como la mejor solución.

Algunas pruebas fueron con otros casos de este problema como el Hamiltoniano donde se regresa de la última ciudad a la primera, sino que solo se recorren todas ellas. En este caso también encontramos al par previamente mencionado y un resultado bastante óptimo.

Conclusiones y recomendaciones

Fue una práctica muy interesante de hacer ya que se podía experimentar bastante con el código del problema de la mochila para ver de qué manera podíamos acercarnos a un resultado más óptimo, añadiendo penalizaciones, aumentando las probabilidades de mutación y posteriormente desarrollar un algoritmo bastante diferente a los vistos para el TSP, ya que este no operaba con bits, sino con el numero de una ciudad durante un recorrido (que bien puede ser considerado como 3 bits por elemento si se requiere), en general, yo considero que esto aportaba bastante a entender un poco más respecto al tema y compararlo un poco con otras formas de resolver los problemas como puede ser el uso de programación dinámica o con un enfoque de fuerza bruta. Algunas recomendaciones para esta práctica son

probar distintos cambios en las probabilidades para mutación, la cantidad de generaciones o la cantidad de cromosomas empleados, ya que ayudan a visualizar la efectividad del algoritmo en algunas ocasiones.

Bibliografía

Garduño Juárez, R. (2018, 21 septiembre). *Algoritmos genéticos*. Conogasi.

http://conogasi.org/articulos/algoritmos-geneticos/

Ma, S. (2020, 2 enero). *Understanding The Travelling Salesman Problem (TSP)*. Routific.

https://blog.routific.com/travelling-salesman-problem

Rosas Trigueros, J. L. (2021). Lab Session 4: Genetic Algorithms for Combinatorial Optimization [Presentación]. Recuperado de IPN ESCOM.