Instituto Politécnico Nacional
Escuela Superior de Cómputo
Algoritmos Bioinspirados
Practica 3 – TSP con AG
Tinoco Videgaray Sergio Ernesto
Jorge Luis Rosas Trigueros
10/11/2022

Marco teórico:

Los algoritmos genéticos nos permiten hallar nuevas soluciones optimas de una forma “aleatoria” basándose en parámetros como el elitismo y el índice de ajuste dado por un “fitness”.

Problema del agente viajero.

También conocido como TSP (Travelling Salesman Problem por sus siglas en ingles), es un problema muy conocido en el mundo de la programación ya que describe una situación en la que un vendedor tiene que viajar por una cantidad determinada de ciudades de tal forma que recorra la distancia mínima entre estas sin pasar por una ciudad mas de una vez. Lo que en teoría de grafos se denomina ciclo Hamiltoniano (ver figura 1.1).

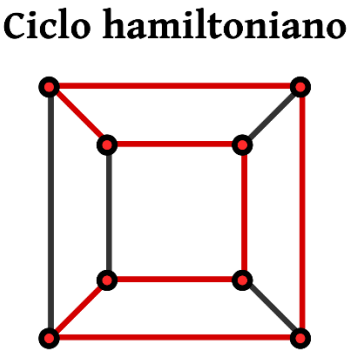

Figura 1.1

Ejemplo de un ciclo hamiltoniano.

Este problema es de tipo no determinista, lo que significa que va a tener múltiples soluciones, de hecho, se estima que va a tener *(n-1)!* cantidad de soluciones, donde n es la cantidad de nodos o ciudades. Esto en el caso de que las distancias sean simétricas, lo que implica que la distancia entre para ir de A hacia B sea distinta que la distancia de B hacia A. Y en el caso de que las distancias sean simétricas entonces la cantidad de soluciones se reduce a la mitad, es decir *(n-1)! /2 soluciones.*

Existen múltiples algoritmos para encontrar soluciones a este problema como lo es el algoritmo de Dijstra para encontrar la distancia más corta, el método del vecino mas cercano, o el método de Branch and bound por mencionar algunos.

Otro posible método para encontrar soluciones es por medio de un algoritmo genético que vaya generando diversas soluciones de forma “aleatoria” dando preferencia a las soluciones más factibles.

En esta practica se va a implementar un algoritmo genético que vaya generando nuevas soluciones para el problema del agente viajero.

Para este problema se tienen 10 ciudades por las que el vendedor debe pasar, es decir se tienen (10-1)! posibles soluciones, lo que da un total de 362880 rutas posibles de las cuales, solo unas pocas serán las más óptimas para este problema.

Las distancias entre las 10 ciudades estarán definidas de acuerdo con la siguiente tabla (ver tabla 1.1).

Ciudad	1	2	3	4	5	6	7	8	9	10	11
1	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	1	2	2	2	2	2	2	2
3	2	2	2	2	1	2	2	2	2	2	2
4	2	2	2	2	2	1	2	2	2	2	2
5	2	2	2	2	2	2	1	2	2	2	2
6	2	2	2	2	2	2	2	1	2	2	2
7	2	2	2	2	2	2	2	2	1	2	2
8	2	2	2	2	2	2	2	2	2	1	2
9	2	2	2	2	2	2	2	2	2	2	1
10	2	2	1	2	2	2	2	2	2	2	2
11	2	2	2	2	2	2	2	2	2	2	2

Tabla 1.1

Matriz de distancias entre las 10 ciudades.

Como se puede observar en la tabla anterior, las distancias entre las ciudades no son simétricas, lo que implica que no es lo mismo viajar de la ciudad 2 a la 4, que viajar de la ciudad 4 a la 2 ya que las distancias difieren.

Material y equipo:

-Equipo: Computadora con conexión a internet.

-IDE: Google Colab.

-Lenguaje de programación: Python 3.6

-Recursos adicionales: Matriz de distancias (tabla 1.1).

Desarrollo de la practica:

TSP con algoritmos genéticos.

Implementación en Python.

Para la implementación de este algoritmo se definió una matriz de distancias entre las 10 ciudades por las que el viajante debe pasar.

```
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
```

Se declaran los vectores para trabajar con los cromosomas, así como el valor del elitismo.

```
cromosomas=[]   #Declarar vector F0
F0=[]   #Vector de ajuste
elitismo=0
```

Se define la función para crear los cromosomas.

```
def crearCromosomas():
  global cromosomas
  cromosomas=[1,2,3,4,5,6,7,8,9,10]
  random.shuffle(cromosomas)  #Aleatorizar lista de cromosomas
```

Se define la función para evaluar la lista de cromosomas por medio del vector de ajuste o “fitness”.

```
def evaluarCromosoma(cromosoma):    #Fitness dado por la distancia recorrida
  F1=[]
  global elitismo
  for g in range(9):
    F1.append(M[cromosoma[g]][cromosoma[g+1]])

  if(F1[0]==1 and F1[1]==1):
    elitismo=3  #Elitismo maximo de 30%
  elif(F1[0]==1):
    elitismo=2  #Elitismo de 20%
  return F1
```

Posteriormente se declara la función para la “mutación” de nuestros cromosomas.

```
def modificarCromosoma(cromosoma,F):    #Insertar ciudad en otra posicion
  global elitismo
  j=0

  c=cromosoma[:]    #Copia de cromosoma
  for i in range(elitismo,8):  #Recorro mi vector de ajuste
    if F[i]==1:  #Si la distancia es minima
      a=c.pop(i)   #Retiro la ciudad A
      b=c.pop(i) #Retiro la ciudad B (i no aumenta porque se quito un elemento)
      c.insert(elitismo,b)    #Los coloco al inicio de la lista
      c.insert(elitismo,a)
      if(j<4):
        j+=2

  genoma=c.pop(random.randrange(elitismo+j,10))   #Retiro un elemento aleatorio
  c.insert(random.randrange(elitismo+j,10),genoma)   #Inserto el elemento en otra po
sicion aleatoria
  return c
```

Se crea la lista de cromosomas de forma aleatoria al inicio del programa.

Posteriormente se evalúa la lista generada y se instancia el vector de ajuste.

```
crearCromosomas()  #Genero el recorrido de ciudades
F0=evaluarCromosoma(cromosomas)
```

Se imprimen los valores de los cromosomas y el vector de ajuste con la matriz de distancias.

```
print("Recorrido: ",cromosomas)
print("Fitness: ",F0)
print("Distancia recorrida: ",sum(F0))
```

Posteriormente se instancia un botón con el método create_button.

Cuando el usuario seleccione el botón, se ejecutará el método nextgeneration.

```
button=create_button()
button.on_click(nextgeneration)
display.display(button)
```

Resultados de la ejecución:

```
Recorrido:  [8, 4, 9, 3, 7, 2, 10, 5, 1, 6]
Fitness:  [2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0]
Distancia recorrida:  18.0
```

Imagen 2.1

Resultados de la primera generación.

```
Recorrido:  [5, 7, 6, 2, 3, 9, 10, 8, 4, 1]
Fitness:  [1.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0]
Distancia recorrida:  17.0
```

Imagen 2.2

Resultados de la segunda generación.

```
Recorrido:  [5, 7, 3, 2, 9, 10, 8, 4, 6, 1]
Fitness:  [1.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0, 1.0, 2.0]
Distancia recorrida:  16.0
```

Imagen 2.3

Resultados de la tercera generación.

```
Recorrido:  [5, 7, 4, 6, 3, 8, 2, 9, 10, 1]
Fitness:  [1.0, 2.0, 1.0, 2.0, 2.0, 2.0, 2.0, 2.0, 2.0]
Distancia recorrida:  16.0
```

Imagen 2.4

Resultados de la cuarta generación.

```
Recorrido:  [5, 7, 8, 10, 4, 6, 1, 9, 3, 2]
Fitness:  [1.0, 2.0, 1.0, 2.0, 1.0, 2.0, 2.0, 2.0, 2.0]
Distancia recorrida:  15.0
```

Imagen 2.5

Resultados de la quinta generación.

```
Recorrido:  [5, 7, 9, 2, 6, 8, 4, 10, 3, 1]
Fitness:  [1.0, 1.0, 1.0, 2.0, 1.0, 2.0, 2.0, 2.0, 2.0]
Distancia recorrida:  14.0
```

Imagen 2.6

Resultados de la sexta generación.

```
Recorrido:  [5, 7, 9, 6, 8, 4, 10, 2, 3, 1]
Fitness:  [1.0, 1.0, 2.0, 1.0, 2.0, 2.0, 2.0, 2.0, 2.0]
Distancia recorrida:  15.0
```

Imagen 2.7

Resultados de la séptima generación.

```
Recorrido:  [5, 7, 9, 2, 4, 6, 8, 1, 10, 3]
Fitness:  [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 2.0, 2.0]
Distancia recorrida:  12.0
```

Imagen 2.8

Resultados de la octava generación.

```
Recorrido:  [5, 7, 9, 2, 8, 4, 6, 10, 1, 3]
Fitness:  [1.0, 1.0, 1.0, 2.0, 2.0, 1.0, 2.0, 2.0, 1.0]
Distancia recorrida:  13.0
```

Imagen 2.9

Resultados de la novena generación.

```
Recorrido:  [5, 7, 9, 2, 4, 1, 3, 8, 10, 6]
Fitness:  [1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 2.0, 1.0, 2.0]
Distancia recorrida:  12.0
```

Imagen 2.10

Resultados de la décima generación.

Tablas:

Ciudad	1	2	3	4	5	6	7	8	9	10	11
1	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	1	2	2	2	2	2	2	2
3	2	2	2	2	1	2	2	2	2	2	2
4	2	2	2	2	2	1	2	2	2	2	2
5	2	2	2	2	2	2	1	2	2	2	2
6	2	2	2	2	2	2	2	1	2	2	2
7	2	2	2	2	2	2	2	2	1	2	2
8	2	2	2	2	2	2	2	2	2	1	2
9	2	2	2	2	2	2	2	2	2	2	1
10	2	2	1	2	2	2	2	2	2	2	2
11	2	2	2	2	2	2	2	2	2	2	2

Tabla 1.1

Matriz de distancias entre las 10 ciudades.

Conclusiones:

En esta practica pude observar con mayor precisión el comportamiento de los algoritmos genéticos, ya que en prácticas pasadas el resultado no era muy superficial y se representaba de una forma mas abstracta considerando los valores de cada cromosoma como una cadena de bits. Lo que contrasta bastante con esta practica en la que se manejaron únicamente valores del 1 al 10 para representar las 10 ciudades y de igual forma los valores del fitness se representaron por los pesos de las distancias entre las ciudades.

De igual forma pude apreciar con mayor claridad cómo funciona la presión selectiva en el caso del elitismo donde se va a dar cierta “preferencia” a los valores colocados al inicio de nuestra solución, los cuales representan un menor peso de la distancia y partir de ahí, modelar una solución que pueda converger a los valores óptimos.

Como practica personal decidí modificar el valor del elitismo máximo para observar el comportamiento del algoritmo con un elitismo menor al 30% y pude apreciar mas aleatoriedad en el orden de los valores, por lo que en algunas ocasiones el resultado del fitness convergía más rápido a un valor cercano a 12 pero de igual forma este valor divergía en 1 o 2 iteraciones, alejándose de un resultado optimo.

Bibliografía:

López, B. S. (2020, 16 febrero). *Problema del agente viajero – TSP*. Ingeniería Industrial Online.

https://www.ingenieriaindustrialonline.com/investigacion-de-operaciones/problema-del-agente-viajero-tsp/

Camino hamiltoniano. (s. f.). http://i3campus.co/CONTENIDOS/wikipedia/content/a/camino_hamiltoniano.html