Documentación de Computación Bioinspirada

Optimización por colonia de hormigas

Autores: Juan Antonio Silva Luján y Luis Bote Gallardo

Universidad de Extremadura, Escuela Politécnica (Cáceres).

Introducción al problema.

Este tipo de algoritmos son tradicionalmente muy conocidos y estudiados en problemas de ámbito de optimización combinatoria computacional, ya que a pesar de su aparente sencillez se trata de problemas muy complejos de resolver.

La dificultad de los problemas de optimización por colonia de hormigas radica en la búsqueda de la solución más eficiente.

Podemos definir este tipo de problemas como una búsqueda eficiente de la ruta más corta que debe llevar a cabo un vendedor que parte desde una ciudad de origen hasta un conjunto de ciudades volviendo, en última instancia, a la ciudad original pero existe una restricción, por cada ciudad solo debe pasar en una única ocasión.

Uno de los métodos más aproximados para resolver este tipo de problemas se basa en el estudio del comportamiento de las colonias de hormigas ya que estas, resuelven el problema de la búsqueda.

Aplicación al problema mediante el comportamiento de las hormigas.

Aunque las hormigas sean ciegas y mantengan un comportamiento casi aleatorio, acaban encontrando el camino más corto desde el punto de origen (nido) hasta el alimento. Además, son capaces de regresar. Todo ello lo logran bajo dos consideraciones:

1. Una sola hormiga es incapaz de llevar a cabo estas acciones, es necesario la colaboración de una colonia.
2. Se hace uso de feromonas para orientar al resto.

El comportamiento del algoritmo podría resumirse a grandes rasgos de la siguiente manera:

Una hormiga (exploradora) se mueve de manera aleatoria alrededor de la colonia. Si esta encuentra una fuente de comida, retorna a la colonia de manera más o menos directa, dejando tras sí un rastro de feromonas. Estas feromonas son atractivas, las hormigas más cercanas se verán atraídas por ellas y seguirán su pista de manera más o menos directa (lo que quiere decir que a veces pueden dejar el rastro), que les lleva a la fuente de comida encontrada por la exploradora.

Al regresar a la colonia con alimentos estas hormigas depositan más feromonas, por lo que fortalezcan las rutas de conexión.

Si existen dos rutas para llegar a la misma fuente de alimentos, en una misma cantidad de tiempo, la ruta más corta será recorrida por más hormigas que la ruta más larga.

En consecuencia, la ruta más corta aumentará en mayor proporción la cantidad de feromonas depositadas y será más atractiva para las siguientes hormigas.

La ruta más larga irá desapareciendo debido a que las feromonas son volátiles (evaporación).

Finalmente, todas las hormigas habrán determinado y escogido el camino más corto.

Implementación del algoritmo en Python

· Evaporación de las feromonas

Mediante la función evaporaferomonas podemos introducir la feromona concreta que deseamos evaporar aplicando un coeficiente del 10% de disminución. La feromona actual se encuentra en uno nodo. Esta técnica es aplicada para evitar la aparición de óptimos locales.

```
#Realizamos la función de ir evaporando las feomonas, se van
multiplicando
#por una constante con valor 0.9 = Coeficiente 10%
def evaporaferomonas(feromonas):

  coeficienteReduccion=0.9
  for lista in feromonas:            #Recorremos la lista de
feromonas, en cada iteración
    for i in range(len(lista)):      #vamos aplicando una reducción
del 10% del valor
      lista[i] = lista[i] * coeficienteReduccion
```

· Rastrear feromonas evaporadas

Durante el proceso de ejecución del algoritmo, cada hormiga es capaz de ir rastreando las feromonas depositadas en las aristas por otras hormigas. Este método ayudará a actualizar estas feromonas por cada hormiga ejecutada.

```
#Actualización de la estructura matricial de feromonas

#dado un camino croncreto

def rastroferomonas(feromonas, camino, dosis):

  for i in range (len(camino) - 1):

    feromonaR = feromonas[camino[i]][camino[i+1]]

    feromonaR = feromonaR + dosis
```

· Hormigas

Implementación del algoritmo del viajante aplicado a la colonia de hormigas. Se recibe por parámetros un array bidimensional.

```
#El método retornará una estructura (tupla) con el camino
#evaluado más prometedor de todos los obtenidos así como
#la distancia total que tiene

def hormigas(matriz, iteraciones, distMedia):

  #Dclaración/Inicialización de matriz para las feromonas
  #por defecto, se inicializará la matriz vacía
  n = len(matriz)
  feromonas = [[0 for i in range(n)] for j in range(n)]

  #Declaración del camino más prometedor y su longitud,
  #inicialmente, se le asignará el máximo valor posible
  #y se irá adaptando a lo largo de la ejecución cuando se
  #vayan actualizando los correspondientes pesos.
  camino_prometedor = []
  distancia_mejor = sys.maxsize

  #Recorremos todas las iteraciones generando nuevas hormigas que
  #irán recorriendo un camino, en caso de tratarse de un camino
  #prometedor que supere al anterior, deposita una feromona que
  #tendrá "mejor" en los casos de que los caminos sean más cortos.
  for iter in range(iteraciones):
    (camino,distancia) = eligecamino(matriz, feromonas)
    if (distancia <= distancia_mejor):
      camino_prometedor = camino
      distancia_mejor = distancia

    dis = distMedia/distancia
    rastroferomonas(feromonas, camino, dis)

    # En cualquier caso, las feromonas se van evaporando
    evaporaferomonas(feromonas)

  # Se devuelve el mejor camino que se haya encontrado
  return (camino_prometedor, distancia)
```

· Elección del nodo

La función de elección del nodo sirve para decidir hacia que nodo se expandirá la hormiga, el método retornará una estructura (lista) con todas las opciones posibles expandibles teniendo en cuenta los valores:

1. Ciudades
2. Feromonas

```
import random, sys, math
#Expande un nodo concreto (camino) analizando los pesos de las
feromonas y
#los nodos que ya han sido visitados anteriormente
def eligenodo(valores, feromonas, visitados):

  #Declaración de valor de cada ciudad
  LValores = []
  LAccesibles = []
  actual = visitados[-1]

  # Declaración de valores de cada ciudadn(alfa -> feromonas &
beta-> valor)
  ALPHA = 1.0
  BETA = 0.5

  #Se recorre la estructura, en cada iteración, se calcula 1.0 + la
feromona
  #correspondiente elevado al valor ALPHA, a continuación, se
calcula el valor
  #del peso mediante la división de 1.0 entre el valor que
corresponde elevado
  #a BETA
  for i in range(len(valores)):
    if i not in visitados:
      ferom = math.pow((1.0 + feromonas[actual][i]), ALPHA)
      ponderacion = math.pow(1.0/valores[actual][i], BETA)*ferom
      LAccesibles.append(i)
      LValores.append(ponderacion)

  #Introducimos en valor una de las opciones de forma aleatoria
pero teniendo en
  # consideración el valor del peso.
  valor = random.random() * sum(LValores)
  acumulador = 0.0
  i = -1
  while valor > acumulador:
    i += 1
    acumulador += LValores[i]
  return LAccesibles[i]
```

· Elección del camino

Este método actúa a partir de la ejecución de la función anterior (eligenodo), usa un nodo retornado por la estructura para decidir cuales son las ciudades más prometedoras para expandir. El método retorna cual será el camino elegido así como su distancia tota.

```
#Recibe una matriz y las feromonas para
def eligecamino(matriz, feromonas):

  #Declaración/Inicialización del camino prometedor y su distancia,
  #por defecto, inicializamos a 0, este valor irá incrementándose
  #a lo largo de la ejecución del algoritmo.
  camino = [0]
  distancia_acum = 0.0

  #Mientras que la estructura que almacena los caminos sea menor
que
  #la matriz, en cada iteración se invoca a la función eligenodo
  #para ir acumulando la distancia según la iteración actual.
  while (len(camino) < len(matriz)):
    nodo = eligenodo(matriz, feromonas, camino)
    distancia_acum = distancia_acum + matriz[camino[-1]][nodo]
    camino.append(nodo)

  #Cuando terminamos con todas las iteraciones anteriores, se
vuelve
  #a ubicar por defecto el nodo 0
  distancia_acum = distancia_acum + matriz [camino[-1]][0]
  camino.append(0)
  return (camino, distancia_acum)
```

· Carga de las ciudades

En el problema tenemos que recorrer distintas ciudades en el menor tiempo posible, a la hora de introducir las ciudades hemos puesto dos opciones:

Matriz aleatoria:

La primera opción es un generador automático, dado un tamaño y un valor distancia máxima

```
#Se crea una matriz de N x N donde N es el número de ciudades
#asignando valores de forma aleatoria
def matrizdistancias(N, distanciaMaxima):
  matriz = [[0 for i in range(N)] for j in range (N)]
  for i in range(N):
    for j in range(i):
```

```
      matriz[i][j] = i+1
      matriz[j][i] = matriz[i][j]
  #Impresión de la matriz resultante
  print("Matriz de ciudades: ")
  print()
  for i in matriz:
    print(i)
  return matriz
```

· Carga por fichero externo

La segunda opción te carga un archivo txt, cuya primera línea son las ciudades, y el resto la matriz con las distancias entre ellas.

```
def matrizdistancias2():
  # Mostramos por pantalla lo que leemos desde el fichero
  print('>>> Cargamos las matriz del txt')
  filename ='/content/ciudadesEjemplo.txt'
  f = open('/content/ciudadesEjemplo.txt')
  # Primera lectura para obviar la cabecera
  ciudades = f.readline()
  print(ciudades)
  # Cargamos los datos
  data = f.read().strip()
  # Se cierra el fichero
  f.close()
  #Separamos los datos en un array
  matriz = [[int(num) for num in line.strip().split(',')] for line
in data.split('\n')]
  for i in matriz:
    print(i)
  return matriz
```

Ejecución de la aplicación

Ejecutamos el método principal que llamará a los métodos anteriores, aquí podemos decidir qué opción utilizar descomentando el método a utilizar

```
from time import time
# Start counting.
start_time = time()
#Declaración de variables y asignación de valores de ejemplo
iteraciones  = 2000
MAXdistancia = 300
############################################################
#Opción 1
ciudades = matrizdistancias(numCiudades, MAXdistancia)
```

```
numCiudades  = 7
################################################################
#Opción 2
#numCiudades = len(ciudades)
#ciudades=matrizdistancias2()
################################################################
#Cálculo de la distancia media -> Cantidad de ciudades multiplicado
#por el valor de la distancia máximo, todo ello partido por 2.
distMedia = numCiudades*MAXdistancia/2
(camino, longCamino) = hormigas(ciudades, iteraciones, distMedia)
#Impresión por pantalla de los valores retornados tras la ejecución
#del programa
print()
print("_____________________________________________________________")
print()
print("Camino -> ", camino)
print()
print("Longitud del camino:    ", longCamino)
print()
# Calculate the elapsed time.
elapsed_time = time() - start_time
print()
print("Tiempo de ejecución: %0.5f segundos." % elapsed_time)
```

Ejemplos de ejecución

Hemos ejecutado varios ejemplos y han sido resueltos satisfactoriamente

El ejemplo más sencillo es el primero, en el que creamos una matriz aleatoria

```
Matriz de ciudades:

[0, 2, 3, 4, 5, 6]
[2, 0, 3, 4, 5, 6]
[3, 3, 0, 4, 5, 6]
[4, 4, 4, 0, 5, 6]
[5, 5, 5, 5, 0, 6]
[6, 6, 6, 6, 6, 0]

_____________________________________________________________

Camino ->  [0, 1, 4, 5, 3, 2, 0]

Longitud del camino:     28.0


Tiempo de ejecución: 0.06002 segundos.
```

El segundo ejemplo es con ciudades y sus distancias reales:

```
>>> Cargamos las matriz del txt
ciudades = ["Badajoz","Caceres","Merida","Plasencia","Malpartida-de-Caceres","Gevora"]

[0, 93, 66, 170, 94, 7]
[93, 0, 75, 80, 11, 85]
[66, 75, 0, 150, 74, 65]
[170, 80, 150, 0, 87, 163]
[94, 11, 74, 87, 0, 88]
[7, 85, 65, 163, 88, 0]

_____________________________________________________

Camino ->  [0, 5, 1, 3, 4, 2, 0]

Longitud del camino:     415.0


Tiempo de ejecución: 0.06165 segundos.
```

Hemos realizado un análisis de ejecución dados distintos tamaños de entrada, es decir, se han modificado la cantidad de ciudades con las que trabajará el algoritmo para analizar la diferencia temporal durante la ejecución.
Ha de tenerse en cuenta que estos tiempos de ejecución se han llevado a cabo en la plataforma de desarrollo Google Colab.

Ciudades	Tiempo
5	0,04306
10	0,05041
20	0,13437
40	1,87673
80	9,34459
160	52,36341

Se pueden observar los resultados ilustrados en la siguiente representación gráfica en la que se representa la variable tiempo en el eje Y y el número de ciudades en el eje X.

Conclusión sobre la práctica

El algoritmo resuelve correctamente el problema en un corto periodo de tiempo, empieza a costarle a partir de las 150 ciudades donde se nota un gran incremento de la variable temporal teniendo en cuenta que sigue una función exponencial.