Algoritmos - Asier Llano - AG3

July 21, 2023

Actividad Guiada 3

Asier Llano

Link repostorio github: [https://github.com/asierllano/03MIAR—Algoritmos-de-Optimizacion—2023]

1 El problema del agente Viajero

Esta actividad guiada proporciona la solución del agente Viajero. Empezamos por la solución propuesta en clase y la vamos optimizando en velocidad y eficacia para proporcionar una solución razonablemente buena de este problema.

[ ]:
```
import tsplib95
import random
from math import e
import copy
import urllib

# Descargamos el fichero de datos (Matriz de distancias)
file = "swiss42.tsp"
urllib.request.urlretrieve("http://comopt.ifi.uni-heidelberg.de/software/
 ↪TSPLIB95/tsp/swiss42.tsp.gz", file + ".gz")
!gzip -f -d swiss42.tsp.gz

# Coordenadas 51-city problem (Christofides/eilon)
#file = "ei151.tsp"; urlib.request.urlretrieve("http://comopt.ifi.
 ↪uni-heidelberg.de/software/TSPLIB95/tsp/ei151.tsp.gz")

# Coordenadas - 48 capitals of the US (Padberg/Rinaldi)
#file = "att48.tsp"; urlib.request.urlretrieve("http://comopt.ifi.
 ↪uni-heidelberg.de/software/TSPLIB95/tsp/att48.tsp.gz")

# Lee el problema del archivo
problema = tsplib95.load(file)

# Nodos & aristas
```

```
nodos = list(problema.get_nodes())
aristas = list(problema.get_edges())
```

2 Funciones generales

[ ]:
```
# Se realiza una permutación aleatoria
def crear_solucion(nodos):
    return random.sample(nodos, k=len(nodos))

# Devuelve la distancia total de una trayectoria / solucion
def distancia_total(problema, solucion):
    distancia_total=0
    for i in range(0, len(solucion)):
        distancia_total += problema.get_weight(solucion[i-1], solucion[i])
    return distancia_total
```

3 Búsqueda aleatoria

[ ]:
```
def busqueda_aleatoria(problema, nodos, n):
    mejor_solucion = None
    mejor_distancia = float("inf")
    for _ in range(n):
        solucion = crear_solucion(nodos)
        distancia = distancia_total(problema, solucion)
        if distancia < mejor_distancia:
            mejor_solucion = solucion
            mejor_distancia = distancia
    return mejor_solucion

# Búsqueda aleatoria con 5000 iteraciones
solucion = busqueda_aleatoria(problema, list(problema.get_nodes()), 500)
print("Mejor solucion", solucion)
print("Distancia", distancia_total(problema, solucion))
```

```
Mejor solucion [0, 32, 37, 3, 14, 31, 8, 25, 9, 23, 29, 40, 22, 1, 5, 35, 15,
36, 33, 12, 11, 21, 41, 13, 26, 2, 38, 10, 6, 24, 39, 27, 30, 19, 4, 28, 34, 17,
7, 20, 18, 16]
Distancia 3956
```

4 Búsqueda local

[ ]:
```
def genera_vecina(problema, solucion):
    mejor_solucion=None
    mejor_distancia=float("inf")
```

```
    for i in range(1,len(solucion)-1):
        for j in range(i+1, len(solucion)):
            vecina = solucion[:i] + [solucion[j]] + solucion[i+1:j] +␣
 ↪[solucion[i]] + solucion[j+1:]
            distancia_vecina = distancia_total(problema, vecina)
            if distancia_vecina <= mejor_distancia:
                mejor_distancia = distancia_vecina
                mejor_solucion = vecina
    return mejor_solucion

print("Distancia Solucion inicial:", distancia_total(problema, solucion))
nueva_solucion = genera_vecina(problema, solucion)
print("Distancia Solucion local:", distancia_total(problema, nueva_solucion))
```

```
Distancia Solucion inicial: 3956
Distancia Solucion local: 3718
```

[ ]:
```
def busqueda_local(problema, solucion):
    distancia=distancia_total(problema, solucion)
    while True:
        vecina = genera_vecina(problema, solucion)
        vdist  = distancia_total(problema, vecina)
        if vdist >= distancia:
            break
        solucion = vecina
        distancia = vdist
    return solucion

print("Distancia Solucion inicial:", distancia_total(problema, solucion))
solucion_local = busqueda_local(problema, solucion)
print("Distanica Solucion local:", distancia_total(problema, solucion_local))
```

```
Distancia Solucion inicial: 3956
Distanica Solucion local: 1798
```

5 Optimización

5.1 Optimización en velocidad de búsqueda

Actualmente el algoritmo tarda 1s (en mi PC) en la búsqueda local, este número es muy mejorable, teniendo en cuenta que la búsqueda local es en una vecindad muy acotada, basada únicamente en el intercambio de nodos. Antes de complicar el algoritmo, ampliando la vecindad, incluyendo multiarranque o. El objetivo no es hacerlo más rápido para buscar más con las mismas técnicas, sino hacerlo más rápido para acelerar el desarrollo y poder introducir después técnicas más complejas.

5.1.1 Optimización en velocidad por representación de pesos más simple

Medimos el rendimiento de la búsqueda local:

[ ]:
```
%%timeit
solucion_local = busqueda_local(problema, solucion)
```

```
1.18 s ± 8.68 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Para mejorarlo cambiamos la función get_weight por una lista de listas (también he probado la solución con matriz de numpy pero como el tamaño no es muy grande, es más eficiente la solución en python puro):

[ ]:
```
# Convierte el problema en una matriz de pesos
def problema_matriz(problema):
    n = len(list(problema.get_nodes()))
    m = [list(range(n)) for _ in range(n)]
    for i in range(n):
        for j in range(n):
            m[i][j] = problema.get_weight(i,j)
    return m

# Devuelve la distancia total de una trayectoria / solucion
def distancia_total(problema, solucion):
    distancia_total=0
    for i in range(0, len(solucion)):
        distancia_total += problema[solucion[i-1]][solucion[i]]
    return distancia_total

# Convertimos el problema en un problema matriz
problema_matriz = problema_matriz(problema)
```

Medimos el nuevo rendimiento:

[ ]:
```
%%timeit
solucion_local = busqueda_local(problema_matriz, solucion)
```

```
69.3 ms ± 2.35 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

En mi PC ha pasado de tardar 1400ms a 81ms (un factor de aceleración de x17). Vamos a poder utilizar este factor de aceleración para poder hacer una búsqueda más extensa, permitiendo una búsqueda local más intensiva o multiarranque.

Comprobamos que ha convergido a una solución de la misma distancia:

[ ]:
```
print("Distanica Solucion local:", distancia_total(problema_matriz,␣
↪solucion_local))
```

```
Distanica Solucion local: 1798
```

5.1.2 Optimización en velocidad evitando generar y calcular todas las vecinas

La mayoría de los enlaces no cambian y se generan listas que solo sirven para recalcular enlaces. Si se utilizan simplemente las diferencias de los enlaces que cambian como figura de mérito, el cálculo

es muchísimo más rápido.

Nótese que como es un caso más complicado, se ha incluido un código de comprobación (comentado en su versión final) para comprobar que las distancias fueron calculadas correctamente.

[ ]:
```
def genera_vecina(problema, solucion):
    mejor_solucion=None
    mejor_distancia=0
    #distancia_orig = distancia_total(problema, solucion)
    for i in range(1,len(solucion)-1):
        ni = solucion[i]     # El punto i
        nip = solucion[i-1]  # El punto i previo
        nin = solucion[i+1]  # El punto i siguiente
        distancia_base = -problema[nip][ni]-problema[ni][nin]
        for j in range(i+1, len(solucion)):
            nj = solucion[j]
            njp = solucion[j-1]
            njn = solucion[j+1 if (j+1) < len(solucion) else␣
 ↪(j+1-len(solucion))]
            distancia_vecina = distancia_base - problema[nj][njn]
            if nin == nj:
                distancia_vecina += problema[nip][nj] + problema[ni][njn] +␣
 ↪problema[nj][ni]
            else:
                distancia_vecina += -problema[njp][nj] + problema[nip][nj] +␣
 ↪problema[nj][nin] + problema[njp][ni] + problema[ni][njn]
            #vecina = solucion[:i] + [nj] + solucion[i+1:j] + [ni] +␣
 ↪solucion[j+1:]
            #assert(distancia_vecina + distancia_orig ==␣
 ↪distancia_total(problema, vecina))
            if distancia_vecina <= mejor_distancia:
                mejor_distancia = distancia_vecina
                mejor_solucion = (i,j)
    if mejor_solucion == None:
        return solucion
    i = mejor_solucion[0]
    j = mejor_solucion[1]
    return solucion[:i] + [solucion[j]] + solucion[i+1:j] + [solucion[i]] +␣
 ↪solucion[j+1:]

print("Distancia Solucion inicial:", distancia_total(problema_matriz, solucion))
nueva_solucion = genera_vecina(problema_matriz, solucion)
print("Distancia Solucion local:", distancia_total(problema_matriz,␣
 ↪nueva_solucion))
```

```
Distancia Solucion inicial: 3956
Distancia Solucion local: 3718
```

[ ]:
```
%%timeit
solucion_local = busqueda_local(problema_matriz, solucion)
```

```
6.84 ms ± 611 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Vemos que con esta nueva búsqueda, el tiempo ha mejorado muchísimo. Hemos pasado de los 1400ms iniciales a 7.5ms (en mi PC), lo que supone una aceleración de x180. Esto nos va a permitir utilizar este tiempo para mejorar el algoritmo buscando vecinos más lejanos (evitando quedarnos en mínimos locales) y proveyendo multiarranque.

[ ]:
```
print("Distanica Solucion local:", distancia_total(problema_matriz,␣
 ↪solucion_local))
```

```
Distanica Solucion local: 1798
```

6 Busqueda con multiarranque

[ ]:
```
def busqueda_multiarranque(problema, n, objetivo=0):
    distancia = float("inf")
    solucion = None
    for _ in range(n):
        s = busqueda_aleatoria(problema, range(0,len(problema)), 1)
        s = busqueda_local(problema, s)
        d = distancia_total(problema, s)
        if d < distancia:
            distancia = d
            solucion = s
            if distancia <= objetivo:
                break
    return solucion
```

[ ]:
```
solucion_multiarranque=busqueda_multiarranque(problema_matriz, 500)
print("Solucion multiarranque", solucion_multiarranque)
print("Distanica Solucion multiarranque:", distancia_total(problema_matriz,␣
 ↪solucion_multiarranque))
```

```
Solucion multiarranque [17, 7, 26, 18, 12, 11, 25, 10, 9, 39, 21, 24, 40, 23,
41, 8, 29, 22, 38, 30, 28, 4, 6, 5, 13, 19, 14, 16, 15, 37, 36, 35, 20, 33, 34,
32, 27, 2, 3, 1, 0, 31]
Distanica Solucion multiarranque: 1487
```

7 Búsqueda de vecinos más amplia

La busqueda de vecinos propuesta en la práctica guiada es de un reemplazo de un vecino por otro. Además, tiene en cuenta que en esa búsqueda de vecinos, además de ser de un único paradigma, no intercambia el nodo 0.

Se plantea una búsqueda de vecinos con el siguiente criterio: Mueve un segmento de n nodos a otra posición al derecho y al revés.

Nótese que el código comentado hace la ejecución mucho más lenta, pero comprueba que las distancias calculadas son las correctas. Para asegurarnos que las distancias calculadas de forma incremental son correctas.

[ ]:
```
def genera_vecina(problema, solucion):
    mejor_solucion=None
    mejor_distancia=0
    # Estrategia: Mueve un segmento de n nodos de la posición i a la posición j
    #distancia_orig = distancia_total(problema, solucion)
    for n in range(1, (len(solucion)-1)//2+1):
        for i in range(len(solucion)):
            # Consigue los nodos de antes, después para coger los dos enlaces
            nif = solucion[i]
            nil = solucion[i+n-1-len(solucion)]
            nip = solucion[i-1]
            nin = solucion[i+n-len(solucion)]
            # Calcula la distancia de eliminar el segmento
            distancia_base = problema[nip][nin] - problema[nif][nip] -␣
↪problema[nil][nin]
            # Ahora mira cual es la mejor posición para insertarlo antes o␣
↪despues
            for jrange in [range(max(0,␣
↪n+i-len(solucion)+1),i),range(i+n+1,len(solucion))]:
                for j in jrange:
                    njp = solucion[j-1]
                    nj = solucion[j]
                    # Calcula la distancia de reintegrar el segmento en otra␣
↪posición
                    distancia_vecina_base = distancia_base - problema[njp][nj]
                    distancia_vecina_directa = distancia_vecina_base +␣
↪problema[njp][nif] + problema[nil][nj]
                    distancia_vecina_inversa = distancia_vecina_base +␣
↪problema[njp][nil] + problema[nif][nj]
                    # Checkea si la medida es mejor que la anterior en directa␣
↪o inversa
                    if distancia_vecina_directa < distancia_vecina_inversa:
                        if distancia_vecina_directa <= mejor_distancia:
                            mejor_distancia = distancia_vecina_directa
                            mejor_solucion = (n,i,j,False)
                    else:
                        if distancia_vecina_inversa <= mejor_distancia:
                            mejor_distancia = distancia_vecina_inversa
                            mejor_solucion = (n,i,j,True)
                    # Genera la vecina y comprueba la distancia
                    #for revierte in [False, True]:
```

```
                        #distancia = distancia_vecina_inversa if revierte else␣
↪distancia_vecina_directa
                        #segmento = solucion[i:min(i+n,len(solucion))] +␣
↪solucion[:max(0,i+n-len(solucion))]
                        #if revierte:
                            #segmento.reverse()
                        #vecina_base = solucion[max(0,i+n-len(solucion)):i] +␣
↪solucion[min(i+n,len(solucion)):]
                        #base = n if j > i+n else max(0, n+i-len(solucion))
                        #vecina = vecina_base[:(j-base)] + segmento +␣
↪vecina_base[(j-base):]
                        #assert(distancia + distancia_orig ==␣
↪distancia_total(problema, vecina))
    if mejor_solucion != None:
        n,i,j,revierte=mejor_solucion
        distancia=mejor_distancia
        # Genera la vecina
        segmento = solucion[i:min(i+n,len(solucion))] + solucion[:
↪max(0,i+n-len(solucion))]
        if revierte:
            segmento.reverse()
        vecina_base = solucion[max(0,i+n-len(solucion)):i] +␣
↪solucion[min(i+n,len(solucion)):]
        base = n if j > i+n else max(0, n+i-len(solucion))
        vecina = vecina_base[:(j-base)] + segmento + vecina_base[(j-base):]
        # comprueba la distancia
        #assert(distancia + distancia_orig == distancia_total(problema, vecina))
        return vecina
    return solucion

print("Distancia Solucion inicial:", distancia_total(problema_matriz, solucion))
nueva_solucion = genera_vecina(problema_matriz, solucion)
print("Distancia Solucion local:", distancia_total(problema_matriz,␣
↪nueva_solucion))
```

```
Distancia Solucion inicial: 3956
Distancia Solucion local: 3619
```

[ ]:
```
%%timeit
solucion_local = busqueda_local(problema_matriz, solucion)
```

```
104 ms ± 1.76 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[ ]:
```
print("Distanica Solucion local:", distancia_total(problema_matriz,␣
↪solucion_local))
```

```
Distanica Solucion local: 1798
```

[ ]:
```
solucion_multiarranque=busqueda_multiarranque(problema_matriz, 100, 1273)
print("Solucion multiarranque", solucion_multiarranque)
print("Distanica Solucion multiarranque:", distancia_total(problema_matriz,␣
 ↪solucion_multiarranque))
```

```
Solucion multiarranque [26, 5, 13, 19, 14, 16, 15, 37, 7, 17, 31, 36, 35, 20,
33, 34, 32, 0, 1, 6, 4, 3, 27, 2, 28, 29, 30, 38, 22, 39, 24, 40, 21, 9, 23, 41,
8, 10, 25, 11, 12, 18]
Distanica Solucion multiarranque: 1273
```

[ ]:
```
%%timeit
solucion_multiarranque=busqueda_multiarranque(problema_matriz, 100, 1273)
```

```
303 ms ± 53.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

8 Conclusiones

Se ha partido de una solución mediocre que en 1400ms obtiene un resultado bastante lejos del mínimo conocido y se ha conseguido obtener el mínimo conocido en 300ms de media.

Para ello se han seguido los siguientes pasos:

- Se ha empezado con el algoritmo heurístico que se había hecho en clase que basándose en una solución aleatoria, la mejoraba buscando un mínimo local, a base de buscar soluciones vecinas. La soluciones vecinas se buscan cambiando un nodo por otro. Era capaz de sacar una solución mínima local en 1400ms.
- Se ha optimizado el funcionamiento de esta solución, precalculando las distancias entre nodos y no recalculando todos los vecinos cuya distancia aumentan. Esto nos ha permitido optimizar el tiempo de una búsqueda local completa y bajarlo de 1400ms a 7 ms.
- Se ha realizado una ejecución multiarranque. Básicamente repite el proceso anterior 500 veces, permitiendo que en 3.4s se obtenga una solución mucho mejor.
- Se ha incorporado un búsqueda de vecinos más amplia y eficaz. La búsqueda de vecinos se realiza intercambiando un conjunto contiguo de n nodos de una posición y colocándolos en otra al derecho y al revés. Esto se ha realiado con la filosofía de computar únicamente los enlaces que cambian. Una búsqueda local de esta forma lleva aproximadamente 100ms.

En una ejecución multiarranque la solución final es capaz de encontrar muy pronto la solución mejor conocida del problema (1273) en unos pocos arranques, tardando tan solo 300ms de media.