Metaheurística
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Práctica 1
Greedy y Busqueda Local en Problema de Asignación Cuadrática(QAP)

Iván Sevillano García

DNI: 77187364-P

E-mail: ivansevillanogarcia@correo.ugr.es

Grupo del martes, 17:30h-19:30h

3 de abril de 2018

Índice

1. Descripción del problema QAP

El problema que se nos plantea es el Problema de Asignación Cuadrática(en adelante QAP por las siglas). En él, tenemos una serie de instalaciones las cuales tienen que interactuar entre ellas una cierta çantidad de trabajo", produciendo un coste. Las instalaciones tienen, además, unas determinadas localizaciones en las que se tienen que situar. Dependiendo de la distancia entre cada dos instalaciones, el coste que producen al interactuar es mayor o menor.

Cabe destacar una serie de detalles en el problema:

- **No Euclideo(No Métrico).** Este problema no pone ninguna objeción a que la distancia de una localización de un lugar a si mismo sea mayor que cero. Además, una instalación puede interactuar consigo misma por consiguiente.
- **No simétrico.** Tampoco pone objeción a que la distancia de una localización a otra no sea la misma que de otra a una.

El problema entonces consiste en repartir las instalaciones en las localizaciones de forma que el coste total o trabajo sea mínimo. Esto es fácil de representar con una permutación, que a cada instalación i le asigna una localización $\pi(i)$. Si llamamos d_{ij} a la distancia que hay de la localización i a la j, y f_{ij} la cantidad de trabajo que tiene que mandar la instalación i a la j, la función de coste asociada al problema sería la siguiente:

$$Coste(\pi) = \sum_{i=1}^{N}\sum_{j=1}^{N} f_{ij}d_{\pi(i)\pi(j)}$$

Si consideramos las matrices de distancias y flujos, D y F y las matrices de que representan a cada permutación y su inversa, Π y Π^{-1} respectivamente, se puede representar el coste de una manera más sencilla:

$$Coste(\pi) =< F, \Pi^{-1}D\Pi >$$

Donde la aplicación $<,>$ tiene como argumentos dos matrices de mismas dimensiones y se aplica en la suma de la multiplicación de sus componentes una a una.

2. Breve descripción de los algorítmos utilizados.

En esta sección vamos a explicar brevemente la configuración de una solución concreta del problema, el funcionamiento de la función de coste y de cómo trabajan los dos algoritmos que se describen en el enunciado.

2.1. Solución.

Una solución está perfectamente determinada por un vector de N componentes donde cada una de sus componentes es distinta unas de otras y el rango de valores difiere de 1 hasta N, o lo que es lo mismo, una permutación. Según la misma, la localización i tendrá alojada la instalación con el número que ocupa en el vector la posición i. Para agilizar cálculos, cada permutación, además, guarda su coste asociado. Así, será fácil calcular soluciones vecinas.

2.2. Función de coste.

La función de coste es la descrita anteriormente en la primera sección.

2.3. Algoritmo Greedy.

Este algoritmo es el más simple que podemos pensar. Éste supone que la localización más cercana a todas las demás debe de ser utilizada por la instalación que más trabajo tenga que realizar, así reduce el "sobrecoste por transporte". Este algoritmo entonces se puede reducir a lo siguiente:

- Ordenar las localizaciones de forma ascendente con respecto a cuanto de lejos está esta de todas las demás.
- Ordenar las instalaciones de forma descendente con respecto a la cantidad de trabajo que deben de realizar en total.
- Asignar por orden las instalaciones y las localizaciones.

2.4. Búsqueda local de primer mejor.

Este algoritmo parte de una solución cualquiera, en nuestro caso utilizaremos la obtenida por el algoritmo Greedy anterior, e intentaremos mejorarla. Para ello, debemos definir y explicar distintos conceptos:

- **Espacio de soluciones.** Podemos abstraer el problema a buscar una permutación dentro de S_N que minimice la función de costo. La existencia es trivial al ser un conjunto finito.
- **Entorno de una solución.** Podemos definir que una solución está çerca"de otra cuando la permutación de una se puede obtener como la otra al aplicarle una

permutación cíclica de orden dos. Así, cada solución tiene $\frac{N(N-1)}{2}$ soluciones cercanas, o vecinos.

- **Factorización del cálculo de coste.** Entre una solución y otra vecina hay muy pocas diferencias, por tanto la función de coste debe de ser fácil de calcular partiendo de una solución vecina base, pues sólo se han visto modificados algunos factores de la función coste. En concreto, sólo en los que intervienen uno o los dos elementos del ciclo disjunto de orden dos explicado anteriormente.

El algoritmo de búsqueda local se basa en encontrar de entre los elementos vecinos de nuestra solución la que sea mejor y quedarnos con esa. Tras esto, repetir con esta nueva mejor solución hasta que nuestra solución sea la mejor de todo su "vecindario". El algoritmo del primer mejor se queda con la primera solución encontrada que mejore a la anterior.

2.4.1. Busqueda local:Don't look Bits

Para agilizar el proceso de busqueda del primer mejor, vamos a desarrollar una estrategia para ahorrarnos calculos. En esta estrategia, vamos a marcar los emplazamientos que no han producido mejoras en búsquedas anteriores y no los vamos a comprobar. Así, nos ahorramos tiempo de cómputo. No nos da la misma solución que la búsqueda local pero nos puede dar una muy buena también y nos ahorra bastante tiempo.

3. Pseudocódigos y explicaciones.

3.1. Algoritmo Greedy

Este algoritmo hace uso de dos funciones distintas. La primera, ordena las filas de una matriz por el valor de la suma de sus valores. No creo necesaria la inclusión de este código ya que cualquier lenguaje de medio nivel tiene una función genérica para ordenar vectores por alguna característica indeterminada.
La segunda simplemente asigna la primera componente del vector ordenado de instalaciones a la última del vector ordenado de localizaciones. Más simple que la función anterior.

3.2. Algoritmo primer mejor: Don't look bits.

3.2.1. Factorización de la busqueda local

Este método devuelve la diferencia de coste de la permutación actual si hubiesemos cambiado la instalación i donde la j y viceversa. Sus variables conocidas son la permutación actual y las matrices de Distancia y Flujo.

```
Parametros:D,F,P
Input:i,j
Output: dif
dif=0
$Modificamos dif con lo que cambian los costes de las localizaciones
$i,j si estuviesen en las localizaciones cambiadas.
dif+=F[i][i]*(D[P(j)][P(j)]-D[P(i)][P(i)])
dif+=F[i][j]*(D[P(j)][P(i)]-D[P(i)][P(j)])
dif+=F[j][i]*(D[P(i)][P(j)]-D[P(j)][P(i)])
dif+=F[j][j]*(D[P(i)][P(i)]-D[P(j)][P(j)])
De k = 1 hasta N con k!=i,j:
  $Aniadimos lo que cambia su coste parcial si la instalacion
  $k tuviese que mandar el flujo a las localizaciones i,j cambiadas.
  dif+=F[k][i]*(D[P(k)][P(j)]-D[P(k)][P(i)])
  dif+=F[k][j]*(D[P(k)][P(i)]-D[P(k)][P(j)])
  dif+=F[i][k]*(D[P(j)][P(k)]-D[P(i)][P(k)])
  dif+=F[j][k]*(D[P(i)][P(k)]-D[P(j)][P(k)])

return difCost
```

3.2.2. Generador de vecino

La forma de generar un vecino dado un ciclo de orden 2 (i,j) es tan simple como intercambiar en la permutación P propia las posiciones i y j y calcular su coste. Puesto que nuestra solución guarda su propio coste y se puede calcular de manera eficiente la

diferencia del coste entre una solución y su vecina, el algoritmo queda así:

```
Parametros:D,F,P,coste
Input:i,j
Output: nuevaP,nuevoCoste

nuevaP = P
nuevaP[i] = P[j],nuevaP[j]=P[i]
nuevoCoste = coste + difCoste(i,j)

return nuevaP, nuevoCoste
```

3.2.3. Generador de soluciones aleatorias

Para generar soluciones aleatorias sólo necesitamos una permutación, la cual la podemos calcular con con el generador aleatorio de python, que tiene una función que mezcla un vector dado, $shuffle()$. Como hemos planteado la implementación de las soluciones, no es necesario calcular ahora mismo el coste que esta permutación tiene.

```
Output: nuevaP
nuevaP = [1..N]\\vector ordenado de 1 hasta N
shuffle(nuevaP)
return nuevaP
```

3.2.4. Exploración del entorno con restricción DLB.

Este método comienza con una solución que es una copia de la solución de arranque y la vamos modificando. Como entrada solo necesitamos un máximo de evaluaciones de la función coste(aunque sólo será necesario llamar a difCoste).

```
Parametros:mejorSol
Input:MaxEval
Output: nuevaP
bitsArray =[0..0]\\vector con N elementos todos 0.
eval=0
Mientras no sobrepasemos MaxEval y mejore en cada iteracion:
  Para cada elemento i de la permutacion:
    Si su bit esta activo:
      exploraVecinos(i)
    Si no ha conseguido mejorar por aqui:
        Cambiamos su bit a 1(inactivo)

return mejorSol
```

La función exploravecinos(i) no es una función que hayamos implementado, lo usamos aquí para que sea más entendible. Aquí el pseudocódigo dentro de la misma:

```
Parametros: mejorSol
Input: MaxEval, i

Para cada j =1..N distinto de i:
  \\Como calculamos difcoste, aniadimos 1 al contador de evaluaciones
  eval+=1
  Si difcoste(i,j) < 0(mejora a la actual):
    Los bits de i y de j se ponen a 0(activos)
    Se reconoce que en esta iteracion han habido mejoras.
    Se cambia mejorSol por este vecino
  Si hemos evaluado ya MaxEval veces la funcion difcoste:
    return mejorSol
```

4. Algoritmo de comparación.

En esta sección se compararán los resultados de los algoritmos Greedy y búsqueda local. Para dicha comparación, atenderemos a dos estadísticos:

- **Desviación a la solución óptima.** Este estadístico evaluará proporcionalmente la diferencia de coste de la mejor solución y la solución obtenida en cada instancia.La fórmula que describe este estadístico es el siguiente:

$$Desv = \frac{1}{|casos|} \sum_{i \in casos} 100 \frac{valorAlg_i - mejorVal_i}{mejorVal_i}$$

- **Tiempo medio de ejecución.** La media de los tiempos de ejecución de cada algoritmo.

Consideraremos entonces que un algoritmo es mejor que otro si la desviación a la solución óptima es menor. En caso de ser iguales, el algoritmo que tarde menos será considerado mejor.

5. Manual de uso.

Para la implementación de la práctica hemos utilizado el lenguaje precompilado Python3, por lo que debe de estar instalado en el sistema. También hemos hecho uso de la biblioteca random del mismo lenguaje(Aunque no para esta práctica, ya que no hemos generado ningún factor aleatorio).

La forma de utilizar el programa es la siguiente:

- **Programa.** El programa recibe como argumento el nombre del archivo de entrada con los datos(terminado en '.dat'), que debe de estar en el directorio *./qapdata/*. Tiene que haber también un archivo solución con el mismo nombre pero con terminación '.sln' en el directorio *./qapsoln/*. Es posible también introducir una semilla como segundo argumento, aunque para este programa no la utilizamos:

  $./Greedy - QAP$ *./qapdata/nombredatos.dat semilla*

  Nos dará como resultado:

  - Solución Greedy. La primera linea será la solución greedy obtenida. La segunda, cuanto tiempo ha tardado el programa en calcularla y cuanto coste total tiene esta solución.
  - Solución BL-DLB. La primera linea será la solución obtenida por busqueda local. La segunda, cuanto tiempo ha tardado el programa en calcularla(incluyendo el cálculo de la primera solución greedy necesaria para comenzar el algoritmo) y cuanto coste total tiene esta solución.
  - Mejor solución. La primera linea es la configuración de la mejor solución y la siguiente es su coste.

- **Experimento total.** Para ejecutar el experimento total, nos hemos ayudado de un script que ejecuta todos los casos de prueba y una makefile que lo llama. Así, la forma de obtener todas las soluciones será tan simple como ejecutar el siguiente comando:

  make greedy

  Todos los resultados nos los encontramos en la carpeta *./solutionGreedy/* con el mismo nombre del archivo de datos pero con terminación ".sol"

6. Experimento y análisis de resultados.

Las tablas obtenidas para cada algoritmo son las siguientes:

Algoritmo Greedy

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	17914	191.0	0.0
chr22b	6194	13510	118.11	0.0
chr25a	3796	18724	393.25	0.0
esc128	64	224	250.0	0.0
had20	6922	7848	13.37	0.0
lipa60b	2520135	3220394	27.78	0.0
lipa80b	7763962	10043715	29.36	0.0
nug28	5166	6770	31.04	0.0
sko81	90998	108158	18.85	0.0
sko90	115534	135404	17.19	0.0
sko100a	152002	178508	17.43	0.0
sko100f	149036	173286	16.27	0.0
tai100a	21052466	24041188	14.19	0.0
tai100b	1185996137	1838134036	54.98	0.0
tai150b	498896643	653315921	30.95	0.0
tai256c	44759294	98685678	120.48	0.0
tho40	240516	338210	40.61	0.0
tho150	8133398	9795960	20.44	0.0
wil50	48816	55760	14.22	0.0
wil100	273038	302048	10.62	0.0

Tabla 6.1: Datos obtenidos para el algoritmo Greedy

Lo más reseñable de este algoritmo es el poco tiempo que se necesita para su ejecución. Esto ocurre por que se puede ordenar vectores en un tiempo $O(nlog(n))$, que es la base de nuestro algoritmo Greedy.

Cabe destacar también la diferencia de desviaciones en distintas instancias. Esto puede llegar a concluir que este algoritmo es muy inestable, ya que la instancia hace variar mucho la calidad de la solución obtenida.

Algoritmo BL

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	6780	10.13	0.1
chr22b	6194	7374	19.05	0.07
chr25a	3796	5900	55.42	0.13
esc128	64	78	21.87	9.98
had20	6922	6984	0.89	0.06
lipa60b	2520135	2520135	0.0	1.79
lipa80b	7763962	9486256	22.18	3.07
nug28	5166	5396	4.45	0.19
sko81	90998	93940	3.23	6.7
sko90	115534	118358	2.44	7.33
sko100a	152002	155784	2.48	12.13
sko100f	149036	151836	1.87	13.7
tai100a	21052466	21907238	4.06	7.22
tai100b	1185996137	1277987724	7.75	12.23
tai150b	498896643	517267383	3.68	32.16
tai256c	44759294	45345150	1.3	55.76
tho40	240516	254682	5.88	0.49
tho150	8133398	8390704	3.16	33.66
wil50	48816	49702	1.81	1.09
wil100	273038	276048	1.1	12.93

Tabla 6.2: Datos obtenidos para el algoritmo BL

Este algoritmo vemos que llega a tardar en algunas instancias hasta un minuto, tiempos muy superiores a los tiempos anteriores(menos de una centésima).

Sin embargo, las desviaciones de este son mucho menores, llegando incluso a alcanzar en la instancia $Lipa60b$ su solución óptima conocida.

Por último, la tabla de comparación de algoritmos:

Algoritmo	Desv	Tiempo
Greedy	71.75	0,00
BL	8.64	11.08

Tabla 6.3: Tabla de comparación

Una vez vistos estos algoritmos, es fácil declarar que el algoritmo de Búsqueda local es mucho mejor a la hora de comparar la desviación del algoritmo. También es verdad, sin embargo, que el algoritmo de búsqueda local parte de una solución dada y la mejora. Como hemos utilizado de solución base la obtenida con el algoritmo Greedy es normal que el segundo se desvíe menos de la mejor solución. Sin embargo, la mejora es bastante notable. Tardar de media 11 segundos en hacer una mejora de alrededor de 60 puntos en el estadístico de Desviación es un intercambio bastante favorable.