Metaheurísticas (2014-2015)
Grado en Ingeniería Informática
Universidad de Granada

Práctica 2 - Búsquedas por trayectorias múltiples

Ignacio Martín Requena

11 de mayo de 2015

Índice

Índice de figuras

1. Descripción del problema

El problema de la asignación cuadrática (QAP) es un problema clásico en teoría de localización. En éste se trata de asignar N unidades a una cantidad N de sitios o localizaciones en donde se considera un costo asociado a cada una de las asignaciones. Este costo dependerá de las distancias y flujo entre unidades, además de un costo adicional por asignar cierta unidad a una localización específica. De este modo se buscará que este costo, en función de la distancia y flujo, sea mínimo.
Este problema tiene muchas aplicaciones, como el diseño de hospitales donde se pretende que los médicos recorran la menor distancia posible dependiendo de su especialidad, procesos de comunicaciones, diseño de teclados de un ordenador, diseño de circuitos eléctricos, diseño de terminarles en aeropuertos y, en general, todo aquel problema de optimización de trayectorias y localizaciones que posea un espacio de búsqueda considerablemente grande.

2. Descripción de los algoritmos empleados

En esta sección vamos a especificar las componentes comunes a todos los algoritmos empleados para la resolución del problema:

- **Representación de la solución:**

  La forma más conveniente considerada para representar las soluciones es a través de las permutaciones de un conjunto. Esto quiere decir que si por ejemplo el problema tiene, por ejemplo, tamaño cuatro, una posible solución al problema sería N = {1,4,2,3}. De esta forma, si interpretamos los índices de este conjunto como las unidades, y el valor del índice como su localización, la localización 3 estaría asignada a la unidad 1, la 4 a la 2 y así sucesivamente.

- **Función objetivo:**

  Dado que el objetivo del problema es la minimización del costo total de todas las posibles soluciones, la función objetivo vendrá definida matemáticamente como:

$$\min_{S \in \prod_N} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} \cdot d_{S(i)S(j)} \right)$$

Donde $\prod_N$ es el conjunto de todas las permutaciones posibles de $N = 1, 2, \ldots, n$

Figura 2.1: Función objetivo

En forma de pseudicódigo nuestra función objetivo sería:

```
inicializamos el costo a 0
para cada fila de las matrices de distancia y flujo
        para cada posicion de las matrices de distancia y
           flujo
                costo += flujo[i][j] * distancia[[i]][sol[j]];
```

- **Función factorizar:**

  Con el fin de aumentar la eficiencia y el tiempo de ejecución de nuestros algoritmos se implementa la función factorizar, que nos calcula la diferencia de costo entre una solución y otra, de esta forma evitamos el tener que calcular el costo de una solución de principio a fin.

  En forma de pseudicódigo nuestra función factorizar sería:

```

    inicializamos una variable suma a 0

    desde i=0 hasta N hacer
            si i no coincide con ninguna de las localizaciones
                a inercambiar
                    realizar el coste del movimiento de
                       intercambio como la sumatoria de la
                       diferencia de todas las distancias
                       nuevas menos las viejas multiplicadas
                       por el flujo
```

- **Función generar vecino:** Esta función calcula una solución vecina a partir de una solución actual y dos posiciones a intercambiar.

  En forma de pseudicódigo nuestra función "swap"sería:

```
1
2  Crear un vector para guardar la nueva solucion
3  Para cada elemento de la solucion actual
4          Copiar en el nuevo vector solucion
5  Guardar en una variable el valor de la posicion a intercambiar
6  Cambiar dicho valor por el contenido en la otra posicion
7  Asignar a la otra posicion el valor guardado en el paso 5
8  Actualizar el costo de la solucion como el costo de la
       solucion inicial mas el factorizado
9  Devolver la nueva solucion
```

- **Función generar solución aleatoria:** Esta función calcula una solución inicial generada aleatoriamente.

  En forma de pseudicódigo nuestra función "getSolAleatoria"sería:

```
1
2  Creamos un vector de enteros con el tamano de las matrices de
       flujo o distancia
3  Para cada posicion del vector creado
4          Generamos un numero aleatorio compremndido entre la
               posicion actual y el tamano del vector
5          Introducimos este numero en el vector de soluciones
               iniciales
6  Calculamos el costo de la solucion inicial
7  Devolvemos la solucion y su costo
```

- **Función BL:** Este algoritmo se compone de dos partes: La creación de la máscara Don't Look Bite y el propio algoritmo de búsqueda

  Una representación en pseudicódigo de esta función sería:

```
1
2  Inicializamos DLB a 0, y con un tamao igual que el del
       problema
3  Creamos una variable para saber cuando parar de iterar
4  Mientras se pueda seguir iterando
5          Para i=1...n
6                  si DLB[i] == 0
7                          Para j=1...n
8                                  si costo factorizado actual
                                       menor que 0
9                                        intercambiamos
                                            localizaciones de
                                            solucion actual
10                                       dlb[i] = dlb[j] = 0;
11                                       paramos de iterar
12                         si podemos seguir iterando
13                                 dlb[i] = 1;
14 Devolver estado actual
```

3. Descripción del esquema de búsqueda y las operaciones de cada algoritmo

3.0.1. Búsqueda Multiarranque Básica

El algoritmo principal para la búsqueda multiarranque básica con el que se ha elavorado la práctica ha sido:

```
1
2  Creamos un vector de soluciones de tam 25
3  Asignamos a cada componente una solucion aleatoria
4  Aplicamos Busqueda local a cada componente
5  Seleccionamos la mejor solucion del vector de tam 25
6  Devolvemos la mejor solucion
```

Aunque su implementación es muy sencilla una vez disponemos del algoritmo de Busqueda Local, el que sea multiarranque proporciona diversidad al explorar soluciones desde muchos caminos diferentes. Los resultados de este algoritmo pese a su simpleza son muy buenos.

3.0.2. GRASP

Esta metaheurística consta de dos partes principales:

- **Función getGreedyAleatorio**

```
1
2           Creamos solucion no inicializada
3
4           Generamos un numero aleatorio entre 0 y el tama del
               problema
5           Creamos dos vectores para gestionar los cambios en la
               solucion, orden y orden_dist
6           Orden dist lo inicializamos con sus componentes
               ordenadas por distancia
7           para i=0 hasta tam
8                   Sumamos los las matrices de flujo a partir del
                        numero aleatorio y lo guardamos en el
                       vector orden
9                   Guardamos el numero de iteracion y se lo
                       asociamos al valor obtenido de la suma de
                       las matrices
10          Ordenamos de menor a mayor el vector orden
11
12
13          solucion[iteracion guardada] = orden_dist
14
15          setCosto(solucion)
16          return solucion
```

- **Algoritmo GRASP**

```
1
2   Creamos dos estados solucion, uno para la actual y otro para
       la mejor
3   Asignamos un valor muy alto al costo de la mejor solucion
4   Mientras i!=25
5                   actual = getGreedyAleatorio();
6                   actual = busquedaLocal(actual);
7                   si actual mejor que la mejoe encontrada
8                           mejor = actual
9                           i = -1
10                  i++
11  Devolver mejor
```

3.0.3. ISL

- **Función mutar**

```
1
2           solucion_mutar = solucion pasada como parametro a la
               funcion
3           int t = tam/4
4           int pos = aleatorio entre 0 y tam-1-tam/4
5
6           para i=0... tam
7                   int auxiliar;
8                   int numero aleatorio 1 = aleatorio entre pos y
                        pos+t
9                   int numero aleatorio 2 = aleatorio entre pos y
                        pos+t
10                  Mientras aleatorio 1 = aleatorio 2
11                          aleatorio 1 = aleatorio entre pos y
                               pos+t
12
13                  aux = solucion mutar[aleatorio1]
14                  solucion_mutar[aleatorio1] = solucion_mutar[
                       aleatorio2]
15                  solucion_mutar[aleatorio2] = aux;
16
17
18          setCosto solucion_mutar
19          Devolver solucion_mutar
```

- **Algoritmo ILS**

```
1
2           Solucion actual = generar aleatoria
3           Mejor solucion = solucion actual
4
5           BusquedaLocal(actual)
6
7           Para i=0 hasta tam
8
9                   actual = mutarcion()
10                  actual = busquedaLocal()
11
12                  si costo actual < costo mejor
13                          mejor = actual
14
15          return mejor
```

4. Algoritmo de comparación: Greedy

Este algoritmo no hace mas que calcular los potenciales de cada unidad y de cada localización, los ordena y asigna el de mayor flujo al de menor distancia.
En pseudocódigo sería algo como:

```
Declaramos un estado solucion
Ordenamos por flujo la matriz de flujo y por distancia la de
   distancia
Para i=0..n
        Asignamos al elemento determinado por flujo del estado
           solucion el elemento distancia (podemos hacerlo asi ya
           que hemos ordenado los vectores previamente.)
Asignamos el costo de la solucion
Devolvemos el estado
```

5. Procedimiento considerado para desarrollar la práctica

Para la realización de esta práctica me he basado en una implementación que he encontrado por internet[1] dado que me ha resultado facil de entender y elegante, sobre todo por la utilización del struc de Estados. Aun así la modificación a este codigo ha sido casi entera y solo se han usado las estructuras y la lectura de los ficheros que el enlace proporciona. Me ha resultado de gran ayuda ya que tenía algunos operadores de copia implementados. El resto de ayudas han venido sobre todo de mano de los apuntes de clase, del guion de prácticas o de charlas con compañeros de clase.

6. Experimentos y análisis de los resultados

Los problemas que he empelado han sido los mismos que los que vienen en la plantilla que se nos proporciona. Para cada caso, los parámetros para su ejecución son:
./qap <Metaheuristica><archivo de entrada><semilla>

donde metaheurística indica la metaheurística a usar:
1) Greedy
2) BMB
3) GRASP
4) ILS

La semilla usada ha sido la **4312365**
A continuación se muestran las tablas con los resultados obtenidos:

[1] http://quadratic-assignation.googlecode.com/svn-history/r44/trunk/qap.cpp

Algoritmo Greedy			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	8916	287,99	0,01
Chr20c	78030	451,76	0,01
Chr22a	13358	116,99	0,01
Chr22b	14620	136,03	0,01
Els19	38627698	124,42	0,01
Esc32b	324	92,86	0,01
Kra30b	117230	28,23	0,01
Lipa90b	16174574	29,50	0,00
Nug30	7410	21,00	0,01
Sko56	40512	17,57	0,01
Sko64	56086	15,65	0,00
Sko72	77296	16,66	0,01
Sko81	102664	12,82	0,00
Sko90	131406	13,74	0,00
Sko100a	176336	16,01	0,00
Sko100b	169670	10,25	0,00
Sko100c	170174	15,09	0,00
Sko100d	171598	14,72	0,00
Sko100e	172328	15,54	0,00
Wil50	49188	0,76	0,01

Figura 6.1: Resultados algoritmo Greedy

Algoritmo BMB			
Caso	**Coste obtenido**	**Desv**	**0,01**
Chr20b	2814	22,45	0,01
Chr20c	19554	38,27	0,01
Chr22a	6530	6,08	0,01
Chr22b	6560	5,91	0,01
Els19	17997928	4,56	0,01
Esc32b	188	11,90	0,03
Kra30b	93030	1,76	0,04
Lipa90b	15176069	21,50	0,85
Nug30	6224	1,63	0,04
Sko56	35038	1,68	0,29
Sko64	49334	1,72	0,44
Sko72	67520	1,91	0,64
Sko81	92520	1,67	0,96
Sko90	117198	1,44	1,33
Sko100a	154024	1,33	1,93
Sko100b	156580	1,75	2,01
Sko100c	149946	1,41	2,01
Sko100d	151458	1,26	1,93
Sko100e	151774	1,76	1,91
Wil50	49188	0,76	0,20

Figura 6.2: Resultados algoritmo BMB

Algoritmo GRASP			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	2884	25,50	0,02
Chr20c	17848	26,21	0,02
Chr22a	6438	4,58	0,02
Chr22b	6790	9,62	0,02
Els19	17997928	4,56	0,01
Esc32b	196	16,67	0,06
Kra30b	93990	2,81	0,04
Lipa90b	15197734	21,67	0,89
Nug30	6216	1,50	0,04
Sko56	35168	2,06	0,27
Sko64	49430	1,92	0,48
Sko72	67534	1,93	1,79
Sko81	92642	1,81	1,30
Sko90	117338	1,56	2,71
Sko100a	154056	1,35	2,12
Sko100b	156024	1,39	3,93
Sko100c	150090	1,51	3,05
Sko100d	152286	1,81	3,09
Sko100e	151452	1,54	3,13
Wil50	49270	0,93	0,29

Figura 6.3: Resultados algoritmo GRASP

Algoritmo ISL			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	2712	18,02	0,01
Chr20c	17634	24,69	0,01
Chr22a	6552	6,43	0,01
Chr22b	6700	8,17	0,01
Els19	18039214	4,80	0,00
Esc32b	192	14,29	0,03
Kra30b	92970	1,70	0,03
Lipa90b	15130341	21,14	2,43
Nug30	6170	0,75	0,04
Sko56	34686	0,66	0,41
Sko64	48828	0,68	0,73
Sko72	66852	0,90	1,19
Sko81	91724	0,80	1,94
Sko90	116302	0,66	3,02
Sko100a	152908	0,60	4,70
Sko100b	154636	0,48	4,71
Sko100c	148548	0,46	4,75
Sko100d	150442	0,58	4,60
Sko100e	150300	0,77	4,71
Wil50	48988	0,35	0,28

Figura 6.4: Resultados algoritmo ILS

	Media Desv	**Tiempo (s)**
GREEDY	**71,8795404704**	**0,00558725**
BMB	**6,5382756623**	**0,7327024**
GRASP	**6,5468143856**	**1,16309325**
ISL	**5,3465859543**	**1,6806456**

Figura 6.5: Comparación de algoritmos

Como podemos ver, el algoritmo que mejor funciona es el ILS, dado que ofrece un poco mas de diversidad que el resto y por tanto evita los óptimos locales. Todos los algoritmos funcionan bien en comparación con el greedy, algunos uncluso se acercan mucho a la mejor solucion, como es el caso de BMB o ILS para wil50.