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

Práctica 4 - Optimización basada en Colonias de Hormigas

Ignacio Martín Requena

8 de junio 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.

2.1. Metaheurísticas y funciones que ayudan al desarrollo de la práctica

- **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 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:

```

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

  Además, la busqueda local usa las siguientes funciones:

- **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:

```
1
2       inicializamos una variable suma a 0
3
4       desde i=0 hasta N hacer
5               si i no coincide con ninguna de las localizaciones
                    a inercambiar
6                       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
```

2.2. Funciones que toman parte en el proceso de contrucción de SCH y SHMM

- **Descripción en pseudocódigo del cálculo de la información heurística:**

```
1
2  para cada elemento de la lista de candidatos
3          incializar variable heuristica
4          para cada elemento de la lista de candidatos
5              HeuristicaC = suma de la distancia de cada par de
                   candidatos
6              heuristica de candidato = 1/HeuristicaC
```

- **Proceso constructivo para generar soluciones:**

  Casi todo el proceso a partir del cual se generan las soluciones se realiza en la función "avanzar" de código adjunto en esta documentación. La descripción de esta función en pseudicódigo es:

```
1
2  Creamos e inicializamos una lista de valores binarios para las
       localizaciones asignadas/sin asignar
3  Buscamos la lista de candidatos, es decir,las localizaciones
     sin asignar (con valor 0)
4  Calculamos la heuristica de cada candidato segun lo descrito
     en el apartado anterior
5  Aplicamos la regla de transicion
6  Lanzamos un valor aleatorio
7  Si es menor o igual que q0
8          Elegir el mejor de los candidatos
9  Si no
10         Hacer ruleta para cada candidato
11         Asignamos la localizacion seleccionada a la hormiga (
             indice de la solucion de la hormiga actual =
             localizacion)
```

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

3.1. SCH

El algoritmo principal el SCH con el que se ha elavorado la práctica ha sido:

```
1
2  Definimos los parametros del algoritmo (alpha, beta, q0...)
3  Ordenamos el vector potencial de flujo de menor a mayor
4  Creamos e inicializamos la matriz de feromona
5  Hasta numero_evaluaciones = MAX_EVALUACIONES
6          Creamos e inicializamos los caminos que van a recorrer las
               hormigas
7
8     //Construimos los caminos de cada hormiga
9      Para cada paso (pasos = tamano problema)
10                Para cada hormiga
11                        Lamar a la funcion avanzar para generar
                             nuevas soluciones
12
13     Calcular el costo de cada camino
14         Hacer busqueda local al mejor camino encontrado
15
16     //Actualizar la feromona
17     Para cada unidad de feromona
18             feromona[i][mejorencontrada->sol[i]] = ((1.0-
                  evaporacion_global)*feromona[i][mejorencontrada->
                  sol[i]]) + (evaporacion_global / mejorencontrada->
                  costo)
19
20
21     Devolver mejor solucion encontrada
```

3.2. SHMM

La metaheuristica SHMM realizada en la práctica en forma de pseudocódigo es:

```
1
2           Generamos una solucion aleatoria
3           Calculamos la matriz de potencial de flujo
4           Concretamos los parametros necearios para el algoritmo (q0
               , evaporacion, alpha, beta, t_max y t_min)
5           Crear e inicializar con t_max la matriz de feromonas
6
7           Mientras no se llegue al maximo de evaluaciones
8                   Creamos la estructura de datos para guardar los
                       caminos de las hormigas
9                   Llamamos a la funcion avanzar para cada hormiga y
                       en cada paso (de esta forma todas las hormigas
                       avanzan a la vez)
10                  Calculamos el costo de cada hormiga
11                  Hacemos una busqueda local a la mejor solucion
                       encontrada
12                  Si la mejor encontrada supera a la mejor hasta
                       ahora nos quedamos con ella y actualizamos los
                       valores de t_max y t_min de la forma: (Los
                       reinicializamos)
13                        t_max = 1.0/(evaporacion_global *
                             mejorencontrada->costo)
14                        t_min = t_max/500.0
15                  //Simulamos la evaporacion de las feromonas
16                  Para cada elemento de feromona
17                          Para cada posicion dentro de feromona
18                                  feromona[i][j] = feromona[i][j]
                                       *(1.0 - coeficiente de
                                       evaporacion global)
19
20                  Buscamos la peor solucion
21
22                  Para cada elemento de feromona en el que este la
                       peor solucion hacer
23                        feromona[i][caminos[idx]->sol[i]] = ((1.0-
                             evaporacion_global)*feromona[i][caminos
                             [idx]->sol[i]]) + (1.0 / caminos[idx]->
                             costo)
24
25                  Si se supera el t_maximo por los decimales,
                       truncar
26
27
28          Devolver la mejor solucion encontrada
```

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.

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

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) SCH
3) SHMM

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

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 SCH			
Caso	**Coste obtenido**	**Desv**	**0,01**
Chr20b	3744	62,92	0,09
Chr20c	50720	258,65	0,07
Chr22a	8092	31,45	0,07
Chr22b	8152	31,61	0,07
Els19	31259342	81,61	0,05
Esc32b	336	100,00	0,09
Kra30b	109360	19,62	0,07
Lipa90b	16257682	30,16	0,18
Nug30	7076	15,55	0,07
Sko56	40798	18,40	0,10
Sko64	57760	19,10	0,12
Sko72	77488	16,95	0,11
Sko81	106348	16,87	0,16
Sko90	135770	17,52	0,11
Sko100a	176406	16,06	0,15
Sko100b	179606	16,71	0,14
Sko100c	173472	17,32	0,14
Sko100d	174872	16,91	0,13
Sko100e	174650	17,10	0,16
Wil50	53732	10,07	0,10

Figura 6.2: Resultados algoritmo SCH

Algoritmo SHMM			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	6746	193,56	0,08
Chr20c	52124	268,58	0,06
Chr22a	9250	50,26	0,06
Chr22b	8998	45,27	0,07
Els19	32706330	90,01	0,06
Esc32b	344	104,76	0,09
Kra30b	113940	24,63	0,08
Lipa90b	16181588	29,55	0,18
Nug30	7308	19,33	0,07
Sko56	40788	18,37	0,11
Sko64	57038	17,61	0,12
Sko72	77292	16,66	0,11
Sko81	106740	17,30	0,16
Sko90	135228	17,05	0,12
Sko100a	176406	16,06	0,15
Sko100b	178942	16,28	0,14
Sko100c	173471	17,32	0,14
Sko100d	174324	16,55	0,14
Sko100e	174650	17,10	0,16
Wil50	53894	10,40	0,10

Figura 6.3: Resultados algoritmo SHMM

Algoritmo	**Desv**	**Tiempo**
Greedy	71,88	0,01
SCH	40,73	0,11
SHMM	50,33	0,11

Figura 6.4: Comparación de los algoritmos

En este caso, aunque pareciera que el proceso busqueda basado en metaheurísticas de colonias de hormigas fuera a dar buenos resultados no ha resultado ser el mejor calculo. Una cosa que aún me inquieta es el hecho de que tarde tan poco en realizar las operaciones (al ser una metaheurística basada en poblaciones el proceso de cálculo debería llevar más tiempo en realizarse, en las tablas no llegan ni a un segundo el tiempo de

calculo). Supongo que aun quedaría mucho por optimizar este método para que diera buenos resultados, por lo que no hay que subestimarlo, seguro que puede dar soluciones mucho mejores que las encontradas actualmente.

A raíz de los resultados tan poco cercanos a los óptimos se me ocurrió modificar la metaheurística para añadirle mas poder de intesificación, es decir, definir un umbral a partir del cual se le aplicara busqueda local a todas las hormigas, pero en vez de mejorar las soluciones las empeoraba así que decidí no incluirlo en la práctica al no aportar nada a la mejora de la solución.