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

Práctica 1 - Búsquedas por trayectorias simples

Ignacio Martín Requena

7 de abril 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 "getSolInicial"sería:

```

Creamos un vector de enteros con el tama\~{n}o de las matrices de flujo o distancia
Para cada posicion del vector creado
        Generamos un numero aleatorio compremndido entre la posicion actual y el tama\~{n}o del vector
        Introducimos este numero en el vector de soluciones iniciales
Calculamos el costo de la solucion inicial
Devolvemos la solucion y su costo
```

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

3.0.1. Búsqueda Local

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

- ***Don't Look Bite:*** Esta técnica permite focalizar la búsqueda en una zona del espacio en la que se puede encontrar una mejor solución. Con la máscara DLB vamos marcando con un 0 las zonas donde nos interesa explorar y con un 1 las que no. El 1 establece que se ha terminado una iteración del bucle interno sin escoger una solución.

Una representación en pseudicódigo sería:

```

Inicializamos DLB a 0, y con un tamao igual que el del
    problema
Para i=1...n
        si DLB[i] == 0
                Para j=1...n
                        vecino = intercambiar(i,j)
                        comparar(vecino, solucion actual)
                        si vecino mejor que solucion actual
                                actual = vecino
                        DLB[i] = DLB[j] = 0
```

- **Búsqueda Local** Ahora mostramos una descripción en pseudocódigo del algoritmo que implementa la búsqueda local:

```

Declaramos una solucion que contendra la solucion vecina a la
    actual
Para cada i=0...n hacer
 Si DLB.at(i) == 0
  Generarvecino()
  Si costo vecino < costo actual
   solucion vecina = solucion actual
   DLB.at(i) = DLB.at(j) = 0
                Si solucion actual = solucion inicial
                        DLB.at(i) = 1
Devolver solucion
```

3.0.2. Enfriamiento simulado

Para la metaheurística de enfriamiento simulado será necesario implementar dos algoritmos:

- **Función temperatura inicial**

  Esta función establece la temperatura inicial a partir de la siguiente fórmula:

$$T_0 = \frac{\mu \cdot C(S_0)}{-\ln(\phi)}$$

Figura 3.1: Temperatura inicial

Donde T0 es la temperatura inicial, C(S0) es el coste de la solución inicial y $\phi \epsilon [0, 1]$ es la probabilidad de aceptar una solución un μ por 1 peor que la inicial.

En forma de pseudocódigo nuestro algoritmo sería:

```
Declaramos una variable de tipo real inicializada a 0 donde
    almacenaremos la temperatura inicial
Aplicamos la formula estipulada anteriormente
Devolvemos el resultado obtenido
```

- **Algoritmo de enfriamiento simulado** Para la realización de este algoritmo tendremos una solución inicial generada aleatoriamente, una solución que será la mejor encontrada hasta el momento y una serie de parámetros que forman parte del algoritmo de enfriamiento simulado tales como la velocidad con la que enfriamos la temperatura (delta), las iteraciones realizadas, todo el espacio de vecindad de una solucion...

  El pseudocódigo del algoritmo que he implementado es el siguiente:

```

Declaramos dos conjuntos de soluciones aleatorias
Inicializamos los parametros ro, delta, temperatura inicial, y
     demas contadores necesarios para controlar la parada del
    algoritmo
Mientras el numero de iteraciones < n*10 o T < 0.001
        Declaramos dos contadores para controlar el descenso
            de la temperatura
        Mientras ambos contadores sean menores que un valor
            inicial establecido
                GenerarVecindario
                Para cada i=1..n y siempre que i < n*(n-1)/2
                        Seleccionamos un vecino
                        Calculamos la diferencia de coste
                            entre la solucion y el vecino
                            generado
                        Definimos la variable de probabilidad
                            de incremento energetico e
                        Si la diferencia es menor que 0 o un
                            numero aleatorio es menor que e
                                Guardamos la solucion en una
                                    variable
                                Si esta solucion es mejor que
                                    la declarada inicialmente
                                        Se elige como mejor
                                            solucion
                                        Se pone el numero de
                                            iteraciones a 0
        Actualizamos la temperatura, las iteraciones y el
            valorestablecido para los contadores
```

```
Devolvemos la solucion mejor
```

3.0.3. Búsqueda tabú

Para la búsqueda tabú en primer lugar debo comentar que me ha sido imposible hacer una implementación buena con multiarranque, por lo que he optado por realizar unn algoritmo de busqueda tabú del mejor de todo los vecinos.
El algoritmo en pseudocódigo es el siguiente:

```

Declaramos e inicializamos como correspondados contadores, una
    lista tabu, una variable para almacenar las vecindades, una
    variable para almacenar la mejor solucion y un conjunto de
    soluciones para almacenar la actual
Inicializamos a NULL cada un de los componentes de la lista tabu
Mientras i < dim*10
        Obtenemos todos los vecinos a partir del actual
        Colocamos un costo muy alto a los estados que se
            encuentran en la lista Tabu comprobando si un estado
            del vecindario y otro de la lista tabu son iguales
        Buscamos la mejor solucion de la vecindad que no este en
            lTabu
        Si costo actual menor que costo anterior
                Copiar solucion actual como mejor solucion y costo
                    actual como mejor costo
                Poner el contador i a 0
        Incrementar contadores
Devolver solucion
```

Esta implementación de la búsqueda tabú se basa unicamente en la intensificación, dado que se vuelve a la mejor solución obtenida hasta el momento y se continua a partir de ella. La ausencia de diversificación puede hacer que caigamos en óptimos locales muy fácilmente, por lo que es probable que los resultados de esta implementación no sean del todo buenos.

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, sobre todo en la forma de leer el archivo dat, el algoritmo de busqueda local, de enfriamiento y el tabu. 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) Busqueda Local
3) Enfriamiento Simulado
4) Busqueda Tabú

La semilla usada ha sido la **28345234**
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**	**Desv**	**Tiempo**		
Chr20b	9956	333,25	0,01		
Chr20c	107970	663,47	0,01		
Chr22a	12888	109,36	0,01	**Media Desv:**	**160,29**
Chr22b	14685	137,08	0,01	**Media Tiempo:**	**0,01**
Els19	14308464	-16,87	0,01		
Esc32b	395	135,12	0,01		
Kra30b	2711025	2865,46	0,01		
Lipa90b	174476	-98,60	0,01		
Nug30	3999	-34,70	0,01		
Sko56	9312	-72,98	0,01		
Sko64	4558	-90,60	0,01		
Sko72	12221	-81,55	0,01		
Sko81	6670	-92,67	0,01		
Sko90	13262	-88,52	0,01		
Sko100a	8548	-94,38	0,01		
Sko100b	8548	-94,45	0,01		
Sko100c	8548	-94,22	0,01		
Sko100d	8548	-94,29	0,01		
Sko100e	8548	-94,27	0,01		
Wil50	53748	10,10	0,01		

Figura 6.1: Resultados Greedy

Algoritmo ES			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	10531	358,27	0,26
Chr20c	99448	603,21	0,53
Chr22a	17022	176,51	0,39
Chr22b	14786	138,71	0,39
Els19	28413657	65,08	2,48
Esc32b	508	202,38	0,24
Kra30b	3135300	3329,56	3,53
Lipa90b	3638489	-70,87	0,26
Nug30	3711	-39,40	0,41
Sko56	10204	-70,39	1,80
Sko64	8242	-83,01	2,25
Sko72	10406	-84,29	3,08
Sko81	7008	-92,30	3,66
Sko90	8458	-92,68	4,73
Sko100a	9016	-94,07	5,98
Sko100b	9016	-94,14	5,94
Sko100c	9016	-93,90	5,95
Sko100d	9016	-93,97	5,95
Sko100e	9016	-93,96	5,93
Wil50	53362	9,31	2,38

Media Desv:	**194,00**
Media Tiempo:	**2,81**

Figura 6.2: Resultados Busqueda local

Algoritmo BL					
Caso	**Coste obtenido**	**Desv**	**Tiempo**		
Chr20b	10531	358,27	0,01		
Chr20c	99448	603,21	0,01		
Chr22a	17022	176,51	0,01	**Media Desv:**	**192,61**
Chr22b	14786	138,71	0,01	**Media Tiempo:**	**0,03**
Els19	28413657	65,08	0,01		
Esc32b	508	202,38	0,01		
Kra30b	3135300	3329,56	0,01		
Lipa90b	165476	-98,68	0,01		
Nug30	3711	-39,40	0,01		
Sko56	10204	-70,39	0,02		
Sko64	8242	-83,01	0,02		
Sko72	10406	-84,29	0,03		
Sko81	7008	-92,30	0,03		
Sko90	8458	-92,68	0,05		
Sko100a	9016	-94,07	0,07		
Sko100b	9016	-94,14	0,06		
Sko100c	9016	-93,90	0,07		
Sko100d	9016	-93,97	0,07		
Sko100e	9016	-93,96	0,07		
Wil50	53362	9,31	0,01		

Figura 6.3: Resultados enfriamiento simulado

Algoritmo TS					
Caso	**Coste obtenido**	**Desv**	**Tiempo**		
Chr20b	10531	358,27	0,40		
Chr20c	99448	603,21	0,41		
Chr22a	17022	176,51	0,05	**Media Desv:**	**193,56**
Chr22b	14786	138,71	0,05	**Media Tiempo:**	**5,83**
Els19	28413657	65,08	0,35		
Esc32b	508	202,38	1,46		
Kra30b	3135300	3329,56	1,23		
Lipa90b	2534482	-79,71	11,57		
Nug30	3711	-39,40	1,21		
Sko56	10204	-70,39	7,15		
Sko64	8242	-83,01	10,72		
Sko72	10406	-84,29	14,84		
Sko81	7008	-92,30	12,57		
Sko90	8458	-92,68	10,27		
Sko100a	9016	-94,07	7,86		
Sko100b	9016	-94,14	8,06		
Sko100c	9016	-93,90	7,48		
Sko100d	9016	-93,97	8,00		
Sko100e	9016	-93,96	7,78		
Wil50	53362	9,31	5,17		

Figura 6.4: Resultados búsqueda tabú

Como se puede observar, estos algoritmos son muy útiles para problemas de gran tamaño, donde la capacidad de cómputo y la forma en que se realizan las búsquedas son cruciales. Aun así, podemos ver que aunque la busqueda tabú obtiene resultados considerablemente buenos para espacios de busqueda grandes, quizá hubiera sido necesaria algún tipo de reinicialización tal y como se sugería en la practica para aprovechar al máximo su potencial.