Metaheurísticas

Práctica 1.a – Búsqueda local y búsqueda tabú

Grupo 3 – Martes 12:30-14:30

43180612K - Abad Vich, David
DAV00004@RED.UJAEN.ES
GRADO EN INGENIERÍA INFORMÁTICA Curso 2014/2015

Índice

1 CONTENIDOS DEL ARCHIVO

2 DESCRIPCIÓN DEL PROBLEMA

De forma general, se trata de encontrar una solución al problema de asignación cuadrática (del inglés, Quadratic Assignation Problem o QAP).

Para este problema, disponemos de *n* unidades y localizaciones, de modo que debemos asignar, de manera óptima, las unidades a las localizaciones. Para ello se dispone de un método para evaluar cómo de óptima es una solución y es mediante el flujo que hay entre diferentes unidades y la distancia que hay entre las localizaciones.

Sin embargo la descripción de encontrar una solución es bastante genérica. Debido a la cantidad de tiempo que nos puede llevar el conseguir la solución más óptima, dependiendo del tamaño del problema, en ocasiones nos tendremos que conformar con encontrar una solución, óptima por supuesto, pero no la más óptima que exista.

Entonces, el coste de una solución se puede formular como:

$$\pi \in \prod_{N}^{min} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} \cdot d_{\pi(i)\pi(j)} \right)$$

donde:

- π es una permutación solución al problema, que representa la asignación de la unidad *i* a la localización π(i).
- f_{ij} es el flujo que hay de la unidad *i* a la *j*.
- d_{kl} es la distancia existente entre la localización *k* y la *l*.

Para este problema se usarán 20 casos diferentes, seleccionados de varios conjuntos de instancias disponibles en la QAPLIB (la biblioteca de problemas de asignación cuadrática) donde se puede encontrar además una descripción de lo que representa el problema.

3 APLICACIÓN DE LOS ALGORITMOS

Al aplicar los algoritmos al problema, hay que tener que tienen en común varios aspectos:

- Representación del problema: El problema se representa como dos matrices, una que contiene los flujos y otra las distancias, de manera que se puede ver el flujo que hay entre dos unidades y la distancia entre dos localizaciones de forma inmediata. Hay que tener en cuenta que mientras que la distancia entre dos localizaciones es la misma en ambos sentidos, puede que el flujo sea distinto entre dos unidades, dependiendo del sentido.

- Representación de las soluciones: Hemos mencionado antes la representación para la solución y es utilizando una permutación de *n* elementos (siendo *n* el tamaño del problema) a modo de que cada posición de la permutación representa una unidad y cada elemento de la permutación representa su localización asignada:

$$solución: \begin{matrix} unidad \\ localización \end{matrix} \begin{pmatrix} 1\ 2 \cdots n \\ i\ \ j \cdots k \end{pmatrix}\ i, j, k \in \{1, \dots, n\}$$

- Función objetivo: La función que se utiliza para calcular el coste de una solución. Su pseudocódigo[1] es:

```
coste ← 0
tamaño ← tamaño(solución)
Para i ← 0 Hasta tamaño-1 Hacer
        Para j ← 0 Hasta tamaño-1 Hacer
                coste ← distancia[solución[i]][solución[j]] * flujo[i][j]
        Fin_Para
Fin_Para
```

- Generación de vecino: Para generar el entorno, se generarán los vecinos mediante el intercambio de dos elementos de la permutación solución.

- Función de factorización: Para un cálculo eficiente del coste de una solución, se extraerá del coste de la solución original los costes relacionados con las dos posiciones que se van a intercambiar para generar esa solución.

- Criterio de parada: Tras evaluar 10000 soluciones distintas, se detendrá la ejecución del algoritmo.

- Generación de la solución inicial aleatoria: En ambos algoritmos se utiliza la misma función para crear una solución inicial aleatoria de la que partir. El pseudocódigo es:

[1] Nota de programación: Hay que tener en cuenta que en el código los vectores y arrays empiezan desde la posición 0, por lo que queda representado en el pseudocódigo.

```
Función generarSolucionAleatoria() Devuelve un vector con la permutación
        tamaño ← tamaño del problema
        Para i ← 0 Hasta tamaño Hacer
                encontrada ← falso
                Mientras (no encontrada) Hacer
                        crear localización aleatoria
                        Para j ← 0 Hasta tamaño Hacer
                                Si (localización está en el vector) Entonces
                                        encontrada ← false
                                Fin_Si
                        Fin_Para
                        vector[i] ← localización
                Fin_Mientras
        Fin_para
        devuelve vector
Fin_Función
```

4 ALGORITMOS UTILIZADOS

4.1 BÚSQUEDA LOCAL

Generamos una solución inicial aleatoria y exploramos el entorno, siguiendo la técnica del primer mejor, por lo que en cuanto encontramos una solución mejor, nos movemos a ella. El entorno se recorre además de forma secuencial con la ayuda de la máscara *don't look bits*, por lo que se exploran sólo las unidades que están activas. Pseudocódigo:

```
Procedimiento buscar()
        Generar solución inicial aleatoria
        tamaño ← tamaño(solución)
        Crear e inicializar array don't look bits dlb
        i ← 0
        mejora ← true
        Mientras (mejora AND soluciones_comprobadas < 10000) Hacer
                j ← i+1
                solución encontrada ← false
                Si (dlb[i] != 0) Entonces
                        //Si el elemento está desactivado, no entra en el siguiente bucle
                        encontrada ← true
                Fin_Si
                Mientras (j < tamaño AND no encontrada) Hacer
                        soluciones_comprobadas++
                        coste ← calcular el coste de cambiar las posiciones i y j
                        Si (coste < coste de la solución) Entonces
                                Se activa el elemento con dlb[i] ← 0
                                Se cambia la solución actual por la encontrada
                                encontrada ← true
                                i ← 0
                        Fin_Si
                        j++
                Fin_Mientras
                Si (no encontrada) Entonces
                        Se desactiva el elemento con dlb[i] ← 1
                Fin_Si
                Si (i == tamaño - 1) Entonces
                        //Si ya ha recorrido todos los elementos, es que no hay mejora
                        mejora ← false
                Fin_Si
                i++
        Fin_Mientras
Fin_Procedimiento
```

En la búsqueda local, además de parar tras comprobar 10000 soluciones, también hay otro criterio de parada y es cuando no hay posibilidad de mejora en el entorno.

No habrá posibilidad de mejora cuando ya se han comprobado todos los cambios posibles, por lo que también la máscara se encontrará a 1 en todos sus bits (todos activados ya que no ha habido mejora para ninguno de los intercambios posibles).

4.2 BÚSQUEDA TABÚ

Se genera una solución inicial aleatoria como en la búsqueda local y exploramos el entorno, en este caso generando 30 soluciones vecinas aleatorias y guardando la mejor. Contamos además con nuestra lista tabú que nos sirve de ayuda para no volver a explorar soluciones que no han sido fructíferas.

Tras la comprobación de esas 30 soluciones, se cuenta una iteración. Cuando pasan 10 iteraciones sin mejora, entonces realizamos una reinicialización. La reinicialización consiste en cambiar la solución de la que exploramos el entorno y lo hacemos de tres formas diferentes, aplicando una de forma aleatoria.

```
Procedimiento buscar()
        Generar solución inicial aleatoria
        tamaño ← tamaño(solución)
        iteraciones ← 0, reinicializar ← true
        Inicializar memoria corto (lista tabú) y largo plazo (matriz de frecuencias)
        Mientras (soluciones_comprobadas < 10000) Hacer
                Para i ← 0 Hasta 30 Hacer
                        Generar r y s posiciones aleatorias
                        Si (intercambio r y s no está en la lista tabú) Hacer
                                soluciones_comprobadas++
                                coste ← calcular coste de cambiar posiciones r y s
                                Si (coste < coste de la solución) Entonces
                                        Se cambia la solución actual por la encontrada
                                        reinicializar ← false
                                        Se actualiza la memoria largo plazo
                                Si_No
                                        Se añade el intercambio r s a la lista tabú
                                Fin_Si
                        Fin_Si
                Fin_Para
                Si (coste solución < coste mejor solución encontrada) Entonces
                        Mejor solución ← solución actual
                Fin_Si
                iteraciones++
                Si (iteraciones == 10) Entonces
                        Si(reinicializar) Entonces
                                Se reinicializa con una de las tres estrategias
                                Se comprueba si va a variar el tamaño de la lista tabú
                                Se reinicia la lista tabú
                        Fin_Si
                        iteraciones ← 0, reinicializar ← true
                Fin_Si
        Fin_Mientras
Fin_Procedimiento
```

Para la reinicialización, el pseudocódigo en los tres casos es:

- Generar una solución inicial aleatoria

```
solución ← generarSolucionAleatoria()
coste ← calcularCoste(solución)
```

- Usar la mejor solución obtenida hasta el momento. Para esto se guarda en una variable de la clase la mejor solución que se ha encontrado hasta el momento, de manera que después de cada iteración, se comprueba si se ha encontrado una mejor solución.

```
solución ← mejor_solucion
coste ← mejor_coste
```

- Usar la memoria a largo plazo para generar una nueva solución

```
solución ← generarSolucionTabu(matriz_frec)
coste ← calcularCoste(solución)
```

La generación de una solución mediante la memoria a largo plazo es:

```
Función generarSolucionTabu(frec)
        tamaño ← tamaño(solución)
        asignado ← vector de booleanos
        Para i ← 0 Hasta tamaño-1 Hacer
                min ← 999999
                Para j ← 0 hasta tamaño-1 Hacer
                        Si(frec[i][j]+ frec[j][i] < min AND no asignado[j]) Entonces
                                min ← frec[i][j]+ frec[j][i]
                                min_pos ← j
                        Fin_Si
                Fin_Para
                nueva_solucion[i] ← min_pos
                asignado[min_pos] ← true
        Fin_Para
        Devolver nueva_solucion
Fin_Funcion
```

5 ALGORITMO DE COMPARACIÓN – GREEDY

Este algoritmo está basado en la búsqueda del primer mejor. Consiste en realizar el sumatorio de cada fila de ambas matrices (flujos y distancias) y asignar a los mayores flujos, las menores distancias.

Es un algoritmo de bajo coste computacional, pues genera una solución (que variará según se asignen las localizaciones a las unidades) que normalmente no llegará a ser un óptimo, pero es probable que se encuentre cerca de uno, por lo que puede tener una gran aplicación si se utiliza para generar soluciones iniciales al problema, de las cuales explorar el entorno para encontrar una mejor solución.

```
Procedimiento CalcularSolucion()
        tamaño ← tamaño(distancias)
        Para i ← 0 Hasta tamaño-1 Hacer
                Para j ← 0 Hasta tamaño-1 Hacer
                        suma_distancias[i] ← suma_distancias[i] + distancias[i][j]
                        suma_flujos[i] ← suma_flujos[i] + flujos[i][j]
                Fin_Para
        Fin_Para
        Para i ← 0 Hasta tamaño-1 Hacer
                Para j ← 0 Hasta tamaño-1 Hacer
                        Si (suma_flujos[i] > max) Entonces
                                Coge nuevo flujo máximo
                        Fin_Si
                        Si (suma_distancias[i] < min AND suma_distancias[i] >= 0) Entonces
                                Coge nueva  distancia mínima
                        Fin_Si
                Fin_Para
                solución[flujo máximo] = distancia mínima
                suma_flujos[flujo máximo] = -1
                suma_distancias[distancia mínima] = -1
        Fin_Para
Fin  Procedimiento
```

Para generar la solución simplemente se suman los flujos y distancias y se asigna al mayor flujo, la menor distancia.

Para evitar volver a coger un elemento que ya ha sido asignado, se le asigna un valor negativo que pasará desapercibido cuando se busquen los siguientes valores máximo y mínimo.

6 PROCEDIMIENTO CONSIDERADO PARA EL DESARROLLO DE LA PRÁCTICA

Para el desarrollo de la práctica se ha utilizado código desarrollado por el propio alumno, con la ayuda de las transparencias de la asignatura y diferentes consultas al profesor.

Además, se han incorporado opciones extra para la práctica que pueden ayudar en el análisis de los resultados, tales como número de iteraciones o la posibilidad de usar la solución del algoritmo Greedy como solución inicial para ambas búsquedas.

6.1 PEQUEÑO MANUAL DE USUSARIO

La ejecución del programa es sencilla:

1. Ejecutar el archivo Meta1a.exe situado en la carpeta Deployment.
2. Elegir el archivo a ejecutar.
3. Introducir los diferentes parámetros:
    a. Matriz que se lee primero.
    b. Valor de la semilla para la generación de números aleatoria.
    c. (Opcional) Usar la solución Greedy como solución inicial de las búsquedas.
    d. (Opcional) Elegir el número de soluciones a comprobar.
4. Hacer clic en ejecutar.

También cabe la posibilidad de ejecutar el programa usando Qt, para ello, una vez instalado Qt, hay que abrir el proyecto que desde la pantalla de bienvenida de Qt, es haciendo clic en la opción Open Project, seleccionando la carpeta donde se encuentra el proyecto y seleccionando el archivo .pro que hay ahí.

Una vez abierto, construir y ejecutar. Saldrá entonces la misma interfaz que con el ejecutable y ya hay que seguir los mismos pasos que con el mismo.

7 EXPERIMENTOS Y ANÁLISIS DE RESULTADOS

Para todos los problemas se ha considerado siempre el mismo valor para la semilla: 123456.

Pasamos a comentar las instancias usadas en esta práctica[2], todas partes de la QAPLIB:

- bur*: Tenemos una matriz que contiene el tiempo medio de escritura de una máquina de escribir y otra con la frecuencia de los pares de letras en diferentes idiomas.
- chr*: Nos dan una matriz de adyacencia de un árbol balanceado y otra matriz con grafos completos.
- els19: Las distancias entre 19 instalaciones diferentes de un hospital y el flujo de pacientes entre ellas. Hay que notar que ésta es de las instancias que tendrá que leerse primero la matriz de distancias.
- esc32a: Ejemplos de una aplicación, usados para circuitos secuenciales auto-comprobables.
- kra32: Contienen datos reales usados para planear el Hospital Regensburg en Alemania.
- lipa90a: Esta instancia proviene de generadores de problemas, que proveen de instancias asimétricas con soluciones óptimas conocidas.
- nug25: La matriz de distancias contiene las distancias de cuadrículas rectangulares de Manhattan. También hay que leer primero la matriz de distancias en esta instancia.
- sko*: Las distancias de éstos problemas son rectangulares y los flujos son número pseudoaleatorios. Otra más para la que se lee primero la matriz de distancias.
- tai*: Instancias uniformemente generadas. Asimétricas y aleatorias, suceden en la generación de patrones de grises.
- tho40: Las distancias de este problema son rectangulares, por lo que la primera matriz que se lee es la de distancias.

Voy entonces con los resultados globales para cada algoritmo, cabe explicar que el tiempo en todos los casos puede variar, puesto que depende de varios factores, sin embargo sigue sirviendo para observar más tarde la media de todos los tiempos, que dará una idea global de la duración de los algoritmos de forma enfrentada.

En cuanto a los costes, hay que comentar también que a la hora de calcula los costes mediante la función de factorización, se han detectado resultados anómalos en dos instancias. Al no poder encontrar una solución a la función de factorización, he seguido usando los resultados anómalos, pues en los otros casos se obtenían resultados válidos.

Todos se han ejecutado también con el estándar de comprobar 10.000 soluciones posibles.

Empiezo con los resultados del algoritmo Greedy:

[2] En caso de no indicar nada, la matriz que se ha de leer primero es la de flujos.

7.1 RESULTADOS GREEDY

Algoritmo Greedy			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Els19	38627698	124,42	865,00
Chr20a	8632	293,80	739,00
Chr25a	17556	362,49	132,00
Nug25	4462	19,18	421,00
Bur26a	5968117	9,98	476,00
Bur26b	4273373	11,93	616,00
Tai30a	2098700	15,43	421,00
Tai30b	1387185541	117,73	590,00
Esc32a	342	163,08	822,00
Kra32	117130	32,05	156,00
Tai35a	2952758	21,91	928,00
Tai35b	506552163	78,79	525,00
Tho40	312578	29,96	136,00
Tai40a	3730054	18,82	534,00
Sko42	18902	19,54	468,00
Sko49	27626	18,13	451,00
Tai50a	5804880	17,54	257,00
Tai50b	788404422	71,83	167,00
Tai60a	8345630	15,82	295,00
Lipa90a	367797	1,99	471,00

Como se puede ver, la desviación es más baja por lo general en archivos generados aleatoriamente, que podría ser coincidencia en el momento de la generación y es que al tomar la suma de los valores de la matriz, al haber sido generada aleatoriamente, los datos entre sí no difieren en gran medida y se pueden sacar soluciones sencillas rápidamente.

Sin embargo en problemas más reales, donde las distancias tienen valores más variados, se puede ver (en concreto con los chr, problemas de árboles) que al generar la solución se han obtenido peores valores, y es que en este caso ya se generará, en la mayoría de los casos, una solución parecida para el mismo problema, ya que las sumas de los valores van a ser más pronunciadas.

Hay que tener en cuenta que al generar la solución Greedy, los resultados pueden variar según cómo se recorra la matriz para obtener los valores o según cómo se traten. En mi caso se recorre secuencialmente y quedándose los primeros mejores valores, por lo que si se encuentran más repetidos, no se tomarán como mejores.

Paso a hablar de la búsqueda local.

7.2 RESULTADOS BÚSQUEDA LOCAL

Algoritmo BL			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Els19	23734354	37,89	865,00
Chr20a	3912	78,47	739,00
Chr25a	8028	111,49	132,00
Nug25	3962	5,82	421,00
Bur26a	5468860	0,78	476,00
Bur26b	3803510	-0,38	622,00
Tai30a	1916134	5,39	421,00
Tai30b	691972163	8,61	590,00
Esc32a	170	30,77	822,00
Kra32	98630	11,20	156,00
Tai35a	2585098	6,73	928,00
Tai35b	331332812	16,95	525,00
Tho40	264608	10,02	136,00
Tai40a	3339928	6,39	534,00
Sko42	16944	7,16	468,00
Sko49	24744	5,81	451,00
Tai50a	5265454	6,61	258,00
Tai50b	487830146	6,32	167,00
Tai60a	7717196	7,09	296,00
Lipa90a	363582	0,82	472,00

Se han mejorado bastante los costes. Como comenté anteriormente, en el Bur26b se encuentra el dato anómalo.

Prestando atención al resto, se han obtenido mejores resultados con una simple búsqueda local, aunque pueden apreciarse algunas instancias en las que la búsqueda local se ha estancado en algún óptimo local y ha dejado un coste más elevado de lo normal, sobre todo en los Chr, los problemas de árboles, de similar forma que al generar la solución Greedy también se obtenían costes más elevados.

7.3 RESULTADOS BÚSQUEDA TABÚ

Algoritmo BT			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Els19	23539778	36,76	867,00
Chr20a	3416	55,84	741,00
Chr25a	5772	52,05	135,00
Nug25	3848	2,78	424,00
Bur26a	5438104	0,21	481,00
Bur26b	3839064	0,56	622,00
Tai30a	1917604	5,47	425,00
Tai30b	543928056	-14,63	596,00
Esc32a	168	29,23	827,00
Kra32	99030	11,65	165,00
Tai35a	2551262	5,34	933,00
Tai35b	290966654	2,70	536,00
Tho40	254708	5,90	150,00
Tai40a	3323910	5,88	544,00
Sko42	16426	3,88	483,00
Sko49	24656	5,43	476,00
Tai50a	5252236	6,35	279,00
Tai50b	491794581	7,19	209,00
Tai60a	7714120	7,05	335,00
Lipa90a	364269	1,01	632,00

Acabo con la búsqueda tabú. Se observa también un dato anómalo en el Tai30b.

Por lo general han mejorado un poco algunos, otros han empeorado con respecto a la búsqueda local y aquí se puede apreciar en los tiempos que todos tardan un poco más. Al ser un problema pequeño, no es algo que afecte mucho al tomar la decisión sobre que algoritmo usar (aunque en el caso del Lipa90a sí puede apreciarse más la diferencia de tiempo). Sin embargo en problemas de mayor complejidad, esto podría ser un detalle bastante importante para la generación de soluciones.

Se han mejorado los resultados del Chr, siguen siendo los mayores valores, pero se ha ganado bastante a cambio de algo más de tiempo que ha llevado la ejecución. Mientras, en las instancias asimétricas se puede observar un ligero empeoramiento con respecto a la búsqueda local. Son casos en los que aplicar la búsqueda tabú no ha sido efectivo, ya que hemos perdido recursos en obtener un resultado que con una búsqueda local hemos mejorado sin todo el gasto de recursos.

7.4 RESULTADOS GLOBALES

Algoritmo	*Desviación*	*Tiempo (en ms.)*
Greedy	72.22	473.50
BL	18.20	473.95
BT	11.53	493.00

En la tabla con los resultados globales se ve más claro la mejora de la búsqueda tabú. Como en esta ocasión además se manejaban ligeros, la diferencia en costes obtenidos y recursos empleados sigue siendo rentable.

Las soluciones del algoritmo Greedy no han sido tan buenas como para decidir quedarse con ellas sin recurrir a un algoritmo de búsqueda.

En futuros problemas con mayores necesidades de recursos, será bastante interesante tomar la decisión de usar un algoritmo más sencillo como la búsqueda local o usar uno más complejo como la búsqueda tabú, tomando el riesgo de llevar mayor tiempo de ejecución. Hay que tener en mente también las necesidades que tenemos y es que si necesitamos una solución rápida, con el simple requisito de obtener una buena solución, podemos contar con la búsqueda local, mientras que en casos con mayores recursos y menor necesidad de una solución rápida, la búsqueda tabú va a dar un mejor resultado.

He probado también a usar la solución Greedy como solución inicial para el problema. Aunque al principio puede parecer una buena idea, en muchos casos provoca un estancamiento en un óptimo local, en especial para la búsqueda local, ya que para la búsqueda tabú siempre tenemos la reinicialización que nos puede ayudar a salir de esos óptimos locales.

En esa última parte entra en juego también el número de soluciones comprobadas, que está directamente relacionado con el número de iteraciones que se van a realizar. Aunque la búsqueda local finalizará al estancarse en un óptimo local (gracias a la lista Don't look bits), la búsqueda tabú seguirá comprobando posibles soluciones, dando más posibilidad a llegar a una mejor solución o incluso al óptimo global del problema.

En los casos probados, multiplicando por 10 el número de soluciones a comprobar ha dado mejores resultados, sin embargo incrementa bastante el coste en recursos, a veces no pudiendo cumplirse y obligando a la ejecución del programa a acabar inmediatamente.