Metaheurísticas (2015-2016)
Grado en Ingeniería Informática
Universidad de Granada

Práctica 5

Algoritmos Híbridos para el Problema de la Asignación Cuadrática

Algoritmos realizados:
AM-(10,1.0),AM-(10,0.1),AM-(10,0.1mej),Greedy.

Nieves Victoria Velásquez Díaz
nievesvvd@correo.ugr.es
DNI:74695481
Grupo 3 - Viernes 17:30-19:30h.

Índice

1. Descripción del problema.

El problema de QAP consiste en ir asignando N unidades diferentes a N localizaciones también distintas. Cada una de las unidades tiene una distancia y un flujo determinado con respecto a las otras unidades dando lugar a que el problema consista en crear una asignación de unidades intentando minimizar, para todas estas, el producto entre la distancia y el flujo. Para la realización de la práctica se han utilizado los algoritmos híbridos para los casos AM-(10,1.0),AM-(10,0.1),AM-(10,0.1mej), apoyándonos en el algoritmo de la búsqueda local y algoritmos genéticos, así como el algoritmo Greedy.

2. Aplicación de los algoritmos.

A la hora de realizar esta práctica nos hemos apoyado en los siguientes componentes para poder desarrollar los algoritmos.

2.1. Esquema de representación empleado.

A la hora de representar la solución nos basamos en la clase "Data", en ella se encuentran almacenadas las dos matrices con las que trabajamos, la matriz de flujo y la de distancia, una variable que almacena el tamaño de las matrices que, al ser cuadradas, solo guardamos una de las dos dimensiones y la usaremos también como límite a la hora de crear los vectores con la solución. Finalmente, también se encuentran todos los métodos necesarios para trabajar con estos datos siempre que nos haga falta. La solución se representa como un vector de la misma dimensión que la longitud de las matrices donde la posición del vector nos indica el flujo que tiene con respecto a su contenido, que nos indica la distancia a la unidad que representa.

2.2. Descripción de la función objetivo.

Todos los algoritmos se basan en la misma función objetivo, "funcionCoste(vector)", definida como:

```
1  INICIO
2    coste:ENTERO = 0
3        PARA i=0 HASTA nDatos do
4          PARA j=0 HASTA nDatos do
5            si i =\= j
6              coste = coste + flujo(i,j)*distancia(i,j)
7            FIN SI
8          FIN PARA
9        FIN PARA
10       DEVOLVER coste
11 FIN
```

Como vemos, la función coste se encarga simplemente de ir multiplicando el valor de la distancia y el flujo para cada elemento y devolver su valor. Basándonos en esta función, podemos ver si una solución es mejor que otra si su coste es menor.

2.3. Algoritmo de búsqueda local.

Para el algoritmo de búsqueda local se ha procedido como se muestra a continuación:

```
1 INICIO
2  S<- valor por parametro
3  MIENTRAS(!optimo AND contador =\= 400) do
4    recorremos la mascara de bit
5      SI mascara[elemento] esta a 0 THEN
6        recorremos el vector de datos
7          SI el elemento de la mascara es distinto a alguno de los del
               vector
8            costeNuevo<-costeMejor+factorizarCoste(solucion, i,j)
9            SI mejora al factorizar
10             S1<-Intercambiar(mascara[elemento],vector[iterador])
11             S<-Mejor(S,S1)
12           FIN SI
13         FIN SI
14     FIN SI
15 DEVOLVER S
16 FIN
```

El algoritmo se encarga de ir viendo con la máscara de bits cuales son los candidatos marcados como válidos para realizar una factorización previa al intercambio con otra posición, siempre que ambas posiciones sean distintas. En caso de que al intercambiar las posiciones ocurra una mejora, hará el cambio entre las posiciones y comprobará cuál es

la mejor solución, si la que teníamos almacenada de antes o la nueva que hemos obtenido.
Para la creación de la lista de candidatos hemos partido de que todos son buenos candidatos, y se han realizado los intercambios y hemos calculado su coste, tras esto, a partir del coste obtenido, mejora, los marcamos como buenos candidatos, y si no, no los marcamos. Mediante este método, mientras creabamos la lista de candidatos hemos ido explorando el vecindario al ir probando las distintas posibilidades.

2.4. Operador de generador de vecino.

El operador de generar vecino (generarSolucion()), se encarga de ir generando valores con los que rellenar el vector, pero para evitar repeticiones nos apoyaremos en un vector secundario llamado apoyo. En este iremos marcando las posiciones de los valores que han ido saliendo y en nuestro vector salida vamos rellenando las posiciones de una en una con elementos no repetidos. Una vez terminada esta primera parte, vemos los valores que aun no se han colocado almacenados en apoyo y vamos incluyéndolos en nuestro vector en las posiciones que nos han ido quedando libres como se ve a continuación.

```
1 INICIO
2  PARA i=0 HASTA tamanio vector DO
3   generamos Valor
4   SI no esta usado THEN
5    S[i]<- valor
6    valor usado
7   FIN SI
8  FIN PARA
9
10 PARA i=0 HASTA tamanio vector DO
11  SI valor no usado
12   buscamos posicion vacia de S
13    S<-valor
14    valor usado
15  FIN SI
16 FIN PARA
17 DEVOLVER S
18 FIN
```

2.5. Factorización.

Para realizar la factorización, se ha modificado el algoritmo de BL de forma que en lugar de realizar directamente un intercambio entre dos posiciones, (una posición y un candidato), antes se ha comprobado mediante una fórmula si el coste de realizar el intercambio más el coste de la mejor solción econtrada es menor que el coste que tenemos hasta el momento, de ser así, realizaremos el intercambio, si no, no lo haremos ahorrando tiempo de cómputo.
La fórmula seguida es la siguiente:

$$\sum_{k=1,k!=r,s}^{n} f_{rk}(d_{\pi(s)\pi(k)} - d_{\pi(r)\pi(k)}) + f_{sk}(d_{\pi(r)\pi(k)} - d_{\pi(s)\pi(k)}) +$$

$$f_{kr}(d_{\pi(k)\pi(s)} - d_{\pi(k)\pi(r)}) + f_{ks}(d_{\pi(k)\pi(r)} - d_{\pi(k)\pi(s)})$$

3. Algoritmos implementados.

Puesto que hemos realizado tres esquemas de búsqueda sobre un mismo algoritmo, algoritmo genético generacional con cruce posicional, en vez de crear tres algoritmos distintos, al algoritmo de partida una vez hemos creado la nueva población y antes de sobreescribir la antigua, realizamos la búsqueda de las tres formas distintas y con la ayuda de un switch-case de forma que tenemos:

3.1. AM-(10,1.0)

Este será el esquema de búsqueda asociado con el tipo **t** que será invocado cada 10 generaciones. En este esquema de búsqueda realizaremos una búsqueda local sobre todos los miembros de la poblacion de forma que:

```
1 PARA i=0 HASTA tamPoblacion DO
2    memetico<-seleccionamosPrimerElemento()
3    busquedaL<-busquedaLocal(memetico)
4    SI memetico.coste < busquedaL.coste THEN
5      poblacionNueva<-Mejor(memetico, busquedaL)
6    FIN SI
```

```
7 FIN PARA
```

Una vez tenemos la población formada, iremos aplicando la búsqueda local a todos sus miembros y seleccionando en cada caso el mejor entre el elemento y el logrado con su búsqueda local.

3.2. AM-(10,0.1)

Este será el esquema de búsqueda asociado con el tipo **s** que será invocado cada 10 generaciones. En este esquema de búsqueda realizaremos una búsqueda local sobre un subconjunto de los miembros de la poblacion de forma que:

```
1 SI probabilidadBL == 0.1 THEN
2   PARA i=0 HASTA tamPoblacion DO
3     memetico<-seleccionamosPrimerElemento()
4     busquedaL<-busquedaLocal(memetico)
5     SI memetico.coste < busquedaL.coste THEN
6       poblacionNueva<-Mejor(memetico, busquedaL)
7     FIN SI
8   FIN PARA
9 FIN SI
```

Una vez tenemos la población formada, iremos aplicando la búsqueda local a todos aquellos miembros que obtenga la probabilidad de aplicarles la búsqueda local y una vez aplicada, seleccionaremos en cada caso el mejor entre el elemento y el logrado con su búsqueda local.

3.3. AM-(10,0.1mej)

Este será el esquema de búsqueda asociado con el tipo **m** que será invocado cada 10 generaciones. En este esquema de búsqueda realizaremos una búsqueda local sobre un subconjunto formado con los 0.1*N mejores miembros de la poblacion, de forma que dado que los tenemos ordenados con una priority_queue tenemos:

```
1 PARA i=0 HASTA tamPoblacion*0.1 DO
2   memetico<-seleccionamosPrimerElemento()
3   busquedaL<-busquedaLocal(memetico)
4   SI memetico.coste < busquedaL.coste THEN
5     poblacionNueva<-Mejor(memetico, busquedaL)
6   FIN SI
7 FIN PARA
```

En este caso hacemos los mismo que en el primero, pero en vez de aplicar la búsqueda local a todos los miembros de la población sólo se lo aplicaremos a los, en este caso, 5 mejores dado que trabajamos con una población de tamaño 50.

4. Descripción del algoritmo de comparación.

El algoritmo de comparación usado ha sido el algoritmo Greedy. Este algoritmo procede como se muestra a continuación:

```
1 INICIO
2  PARA i=0 HASTA tamanioProblema do
3     PARA j=0 HASTA tamanioProblema do
4        f <- f+MatrizFlujo[i][j]
5        d <- d+MatrizDistancia[i][j]
6     FIN PARA
7     Vflujo[i]<-f
8     Vdistancia[i]<-d
9     f = d = 0
10  FIN PARA
11  ordenarVector(Vflujo)
12  ordenarVector(Vdistancia)
13  PARA i=0 HASTA tamanio_Vflujo do
14         solucion[Vflujo]<-Vdistancia[i]
15  FIN PARA
16  DEVOLVER solucion
17 FIN
```

Este algoritmo, en primer lugar, prepara los vectores de Vflujo y Vdistancia sumando todos los elementos de la matriz de flujo y distancia y los almacenandolos en los vectores Vflujo y Vdistancia respectivamente, tras esto, ordenamos el vector de flujo de mayor a menor y el de distancia de menor a mayor y procedemos a construir la solución, esta se irá construyendo de forma que en cada iteración del bucle que recorre Vflujo, tomaremos el elemento de mayor flujo y lo asociaremos a la unidad de menor distancia hasta tener todas las unidades establecidas.

5. Procedimiento considerado y manual de usuario.

5.1. Procedimiento considerado

El procedimiento generado a la hora de realizar esta practica ha sido el de programarlo siguiendo el guión de prácticas proporcionado en clase, así como el seminario para establecer algunos detalles y el de teoría. La practica ha sido realizada en C++ con ayuda de la STL para poder trabajar con las estructuras de datos de forma más cómoda.

5.2. Manual de usuario.

A la hora de trabajar con esta práctica hay dos formas diferentes de hacerlo:

- Desde terminal: mediante este método ejecutaremos los archivos de uno en uno, de forma que nos posicionaremos en el directorio software y escribiremos en la terminal:

```
1    ./bin/main ./bin/datos/fichero.dat tipo
```

  Donde fichero es el nombre del fichero de datos que queremos ejecutar y tipo será **t,s,m** para especificar esquema de búsqueda especificado siendo estos: **AM-(10,1.0),AM-(10,0.1),AM-(10,0.1mej)** respectivamente. En caso de querer cambiar la seed, se la pasaremos como el cuarto argumento a la hora de ejecutar, quedando en la terminal:

```
1    ./bin/main ./bin/datos/fichero.dat tipo semilla
```

  Donde "semilla" será el valor de la semilla que vamos a usar.

- Desde un bash: existe la posibilidad de realizar una ejecución de todos los ficheros sin necesidad de escribirlo, para ello deberemos, situados en la carpeta FUENTES, ejecutar el script script.sh que se encargará de ir llamando al programa con un fichero distinto cada vez y especificando los tres tipos además del algoritmo Greedy. Para ello escribiremos en la terminal:

```
./script.sh
```

6. Experimentos y análisis de resultados.

A continuación procederemos a comentar los resultados obtenidos para los distintos ficheros y con los distintos algoritmos implementados a partir de la semilla **75654258**

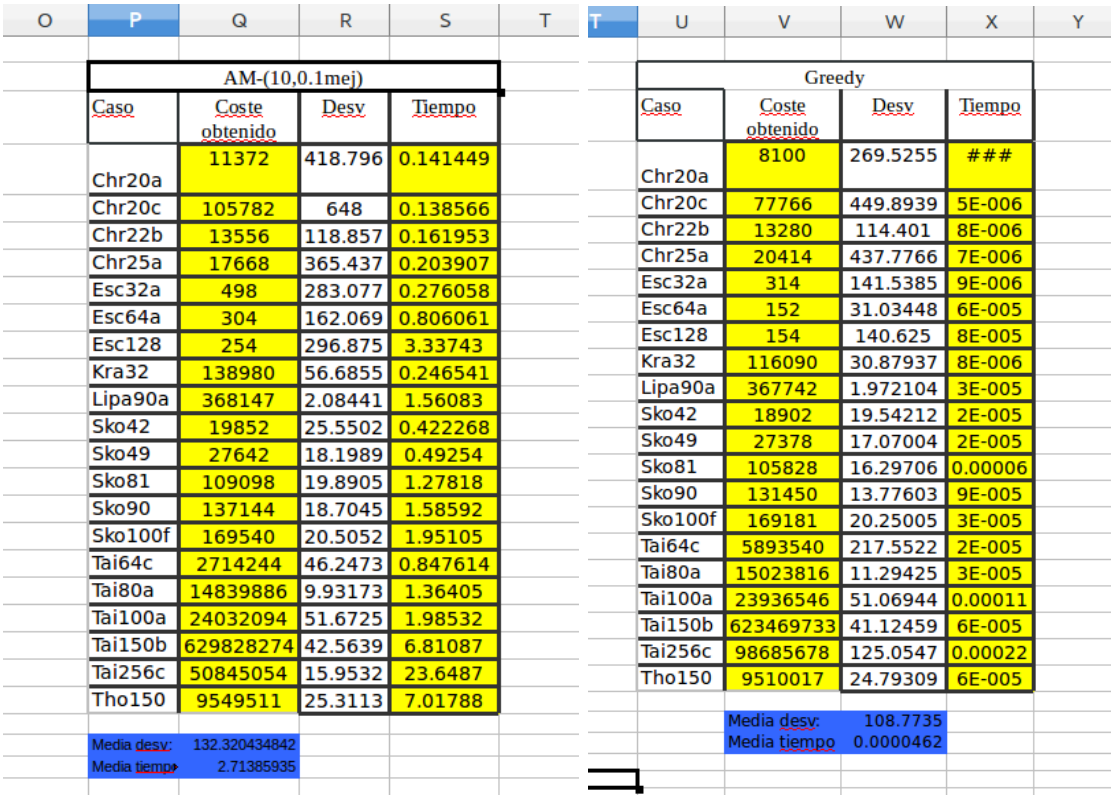
AM-(10,1.0)			
Caso	Coste obtenido	Desv	Tiempo
Chr20a	11372	418.796	0.22994
Chr20c	105782	648	0.20964
Chr22b	13556	118.857	0.24136
Chr25a	17668	365.437	0.32411
Esc32a	498	283.077	0.52597
Esc64a	304	162.069	2.75219
Esc128	254	296.875	17.3949
Kra32	138980	56.6855	0.52297
Lipa90a	368147	2.08441	6.64086
Sko42	19852	25.5502	0.9492
Sko49	27642	18.1989	1.38707
Sko81	109098	19.8905	4.94904
Sko90	137144	18.7045	6.64635
Sko100f	169540	20.5052	8.94881
Tai64c	2714244	46.2473	2.93644
Tai80a	14839886	9.93173	5.01426
Tai100a	24032094	51.6725	11.3113
Tai150b	629828274	42.5639	27.4413
Tai256c	50845054	15.9532	188.685
Tho150	9549511	25.3113	35.1091
Media desv:	132.320434842		
Media tiempo	16.1109898		

(a)

AM-(10,0.1)			
Caso	Coste obtenido	Desv	Tiempo
Chr20a	11372	418.796	0.13328
Chr20c	105782	648	0.13239
Chr22b	13556	118.857	0.15026
Chr25a	17668	365.437	0.16414
Esc32a	498	283.077	0.23107
Esc64a	304	162.069	0.60033
Esc128	260	306.25	1.79639
Kra32	138980	56.6855	0.24415
Lipa90a	368147	2.08441	1.00815
Sko42	19852	25.5502	0.3234
Sko49	27840	19.0456	0.40149
Sko81	109098	19.8905	0.83676
Sko90	137144	18.7045	1.01772
Sko100f	169637	20.5742	1.16283
Tai64c	2714244	46.2473	0.63488
Tai80a	14839886	9.93173	0.93769
Tai100a	24032094	51.6725	1.27186
Tai150b	629828274	42.5639	2.75194
Tai256c	51117316	16.5741	7.15964
Tho150	9550114	25.3193	2.6024
Media desv:	132.866406		
Media tiempo	1.1780386		

(b)

Figura 6.1: Comparacion Algoritmos.



AM-(10,0.1mej)

Caso	Coste obtenido	Desv	Tiempo
Chr20a	11372	418.796	0.141449
Chr20c	105782	648	0.138566
Chr22b	13556	118.857	0.161953
Chr25a	17668	365.437	0.203907
Esc32a	498	283.077	0.276058
Esc64a	304	162.069	0.806061
Esc128	254	296.875	3.33743
Kra32	138980	56.6855	0.246541
Lipa90a	368147	2.08441	1.56083
Sko42	19852	25.5502	0.422268
Sko49	27642	18.1989	0.49254
Sko81	109098	19.8905	1.27818
Sko90	137144	18.7045	1.58592
Sko100f	169540	20.5052	1.95105
Tai64c	2714244	46.2473	0.847614
Tai80a	14839886	9.93173	1.36405
Tai100a	24032094	51.6725	1.98532
Tai150b	629828274	42.5639	6.81087
Tai256c	50845054	15.9532	23.6487
Tho150	9549511	25.3113	7.01788
Media desv:	132.320434842		
Media tiempo	2.71385935		

(a)

Greedy

Caso	Coste obtenido	Desv	Tiempo
Chr20a	8100	269.5255	###
Chr20c	77766	449.8939	5E-006
Chr22b	13280	114.401	8E-006
Chr25a	20414	437.7766	7E-006
Esc32a	314	141.5385	9E-006
Esc64a	152	31.03448	6E-005
Esc128	154	140.625	8E-005
Kra32	116090	30.87937	8E-006
Lipa90a	367742	1.972104	3E-005
Sko42	18902	19.54212	2E-005
Sko49	27378	17.07004	2E-005
Sko81	105828	16.29706	0.00006
Sko90	131450	13.77603	9E-005
Sko100f	169181	20.25005	3E-005
Tai64c	5893540	217.5522	2E-005
Tai80a	15023816	11.29425	3E-005
Tai100a	23936546	51.06944	0.00011
Tai150b	623469733	41.12459	6E-005
Tai256c	98685678	125.0547	0.00022
Tho150	9510017	24.79309	6E-005
Media desv:	108.7735		
Media tiempo	0.0000462		

(b)

Figura 6.2: Comparacion Algoritmos.

Si recogemos los datos de la desviación y el tiempo medio que tarda cada uno obtenemos:

Algoritmo	Desviacion	Tiempo
Greedy	108,7735	0.0000462
AM-(10,1.0)	132,320	16,111
AM-(10,0.1)	132,86	1,178
AM-(10,0.1mej)	132,320	2,713

Como podemos ver, el algoritmo que haya buenas soluciones en un corto periodo de tiempo y el que tiene una menor desviación media es el algoritmo greedy dando lugar a que sea el mejor de los cuatro. Si estudiamos los otros tres algoritmos, podemos ver que el que más

tiempo toma en llevar a cabo su ejecución es el algoritmo AM-(10,1.0). Esto se debe a que aplicao una búsqueda local a todos los miembros de la población y, debido a que se tiene como máximo unas 400 comprobaciones de la funcion de coste para llevar a cabo la búsqueda, junto lo que necesita el algoritmo genético hace que necesite un tiempo superior al de los otros dos algoritmos.
De los otros dos algoritmos, podemos ver que el que aplica el esquema de búsqueda AM-(10,0.1) tiene una desviación algo peor que los otros dos pero apenas perceptible mientras que en tiempo es, en general, el más rápido. Esto puede deberse a que aplica una búsqueda local pero no sobre un conjunto tan grande como puede ser toda la población si no sobre algunos miembros dando lugar a que se agilice la búsqueda mejorando algunos casos. Algo similar ocurre para el caso AM-(10,0.1mej) que en vez de hacerla sobre miembros aleatorios aplica la búsqueda local sobre los 0.1*N mejores intensificando dichos miembros de la pobalción para su posterior torneo y cruce, pero sin intensificar demasiado dado que en el torneo participa la población completa evitando alcanzar un óptimo local demasiado deprisa.
Si comparamos los datos obtenidos en esta práctica con los obtenidos en la tercera práctica podemos ver que se ha obtenido un descanso considerable del tiempo aunque en la mayoría de casos se llegue a una misma solución. Un ejemplo de esto es el caso Sko100f para AM-(10,1.0) que mediante un algoritmo genético necesitaba 1520,95 segundos para hallar una solución y aplicando el algormitmo híbrido se llega a la misma solución en 8,9 segundos. Esto se debe a que, al aplicar una búsqueda local sobre algunos o todos los miembros de la población estos se ven mejorados rápidamente, evitando varios ciclos de los algoritmos genéticos que mecesitan más tiempo computacional que la búsqueda local, además al realiza la factorización se han visto mejorados los tiempos de los tres algoritmos en general pero sobre todo los tiempos del AM(10,1.0) dado que aplica una búsqueda local muy extensa sobre muchos elementos, reduciendo el tiempo que tarda en encontrar los mejores vecinos.

En general podríamos decir que el algoritmo con una mejor relación buenas soluciones/tiempo estaría entre el AM(10,0.1) y AM(10,0.1mej) dado que realizan búsquedas locales sobre subconjuntos aportando una intensificación más rápida pero sin probar todos los casos posibles (lo que hace que el AM(10,1.0) quede descartado por este motivo). Dependiendo de la cantidad de miembros que superen la cota de probabilidad de apicar la búsqueda local podremos decidir cúal de los dos resulta más eficiente aunque en este caso y para los datos obtenidos, el mejor de los dos sería AM-(10,0.1) que mejora algunos miembros de la población aunque no necesariamente sean los mejores dando lugar a que se potencien soluciones que, de primeras, parecían peores aportando algo más de diversidad que su contrincante.
En caso de que no se vean bien las tablas de resultados, estas se encuentran al completo en el documento resultados.xls adjunto en el zip junto al resto de ficheros de la entrega.