Práctica 2 Metaheurísticas QAP

2016/2017

Algoritmos: Greedy, BL, ES, BMB, GRASP, ILS, ILS-ES

Antonio Manuel Milán Jiménez 77449165T

antoniomj@correo.ugr.es

Grupo 3 (Lunes)

Índice

Descripción del problema

El problema de la asignación cuadrática, QAP, nos presenta el problema de determinar la asignación óptima de una serie de unidades en una serie de localizaciones que tenga un costo mínimo. Se nos proporciona las distancias entre las localizaciones y los flujos entre las unidades. Con esta información, se nos pide construir diferentes algoritmos con diferentes técnicas con el fin de obtener buenas soluciones en un tiempo razonable.

Los algoritmos que hemos construido para resolver el problema han sido un algoritmo "greedy", muy rápido pero con mala soluciones como veremos más adelante; una búsqueda local con técnica de "primero el mejor"; el algoritmo de enfriamiento simulado, un algoritmo de multiarranque básico, BMB; un algoritmo GRASP; una búsqueda local reiterada y finalmente una hibridación entre esta búsqueda local reiterada y el algoritmo de enfriamiento simulado.

Representación de las soluciones

Una solución al problema se representa como una permutación de un vector desde el 1 hasta el número de unidades. Así, en cada uno de estos vectores, soluciones, la posición del vector se corresponde con el número de la unidad y el contenido de dicha posición en el vector es la localización.

La valoración de una solución se hace a partir de las matrices de distancia y flujo de cada caso, que nos indica la distancia entre todas las localizaciones y el flujo entre todas las unidades. Dado que intentamos minimizar este valor, diremos que una solución es mejor que otra cuando su valoración sea menor.

Función objetivo

Este es el pseudocódigo de la función objetivo que se encarga de calcular la valoración de una solución a partir de las matrices cómo hemos descrito anteriormente:

```
(for i ← 0…nºunidades)
        (for j ← 0…nºunidades)
                a=solución[i]-1
                b=solución[j]-1
                valor += flujo[i][j] + distancia[a][b]
```

Generación de soluciones aleatorias

```
solucion ← [1…nºunidades]
for(i ← 0…nºunidades)
        r=randInt(0,nºunidades-1)
        Intercambio(solucion, i, r)
```

Búsqueda en enfriamiento simulado

```
while(temperatura_actual>temperatura_final)
        (for i ← 0…max_vecinos)
                a=randInt(0,tam-1)
                b=randInt(0,tam-1) // a!=b
                vecino=Intercambio(solucion,a,b)
                valorVecino=evaluacionFactorial(solucion,a,b)
                if(valorVecino < valorSolucion or U(0,1) <= exp(-dif(f)/k*T))
                        solucion=vecino
                        if(valorSolucion < valorMejorSolucion)
                                mejorSolucion=solucion
        temperatura=enfria(temperatura)
```

Cálculo de temperatura inicial y esquema de enfriamiento

La temperatura inicial se calcula mediante la fórmula:

$$T_o = \frac{\mu * C(S_o)}{-\ln(\emptyset)}$$ donde μ=ø=0.3

Para el esquema de enfriamiento inicialmente probamos con la expresión:

$$temperatura = \frac{temperatura}{1 + \beta * temperatura}$$

Sin embargo, este esquema de enfriamiento hacía que la temperatura descendiese demasiado rápido para poder encontrar una buena solución. Por lo tanto se ha optado por ese otro esquema:

$$temperatura = temperatura * 0.85$$

Búsqueda en BMB

```
for(i ← 0…25)
        solucion=randSolucion()
        solucion = BusquedaLocal(solucion)
        if(i==0 or valorSolucion < valorMejorSolucion)
                mejorSolucion=solucion
```

Búsqueda en ILS

```
solucion=randSolucion()
solucion=BusquedaLocal(solucion)
mejorSolucion=solucion
for(i ← 0…24)
        solucion_aux=mutar(solucion)
        solucion_aux=BusquedaLocal(solucion_aux)
        if(valorSolucion_aux < valorSolucion)
                solucion=solucion_aux
                if(valorSolucion < valorMejorSolucion)
                        mejorSolucion = solucion
```

Operador de mutación en ILS

```
ini=randInt(0,tam-tam/4)
fin=ini+tam/4
subLista = solucion[ini,fin]
subLista = Barajar(subLista)
solucion[ini…fin] = subLista
return solucion
```

Búsqueda en GRASP

```
lcU ← solucionAleatoria
lcL ← solucionAleatoria

cota_u= tam- alpha*(tam-1)
cota_l = 1 +alpha*(tam-1)

lcrU ← lcU(if lcU[i] > cota_u)
lcrL ← lcL(if lcL[i] < cota_l)
```

obtenemos aleatoriamente 2 posiciones y 2 valores que construyen inicialmente la solucion.

```
for(i ← 0…tam)
   if(solucion[i]==0)  //No tiene todavia ningun valor
         for(j ← 0…tam)
            if(!cogido(j))   //Si ese valor no es ya parte de nuestra solucion
                  solucion_aux ← solucion
                  solucion_aux[i] = j+1
                  v = evaluar(solucion_aux)
                  if(v < v_minimo)
                                    v_minimo = v
                              mejor_pareja = pareja(I,j+1)
         lc.add(mejor_pareja)
for(0…lc.tam)
solucion.set(mejor_pareja.pos,mejor_pareja.valor)
```

Búsqueda local:

Operador de generación de vecino:

Para crear un vecino en la búsqueda local o simular una mutación en el genético, básicamente seleccionaremos dos unidades y las intercambiaremos. El qué unidades seleccionar lo describiremos más adelante con el esquema de exploración y su máscara de "Don´t look bits" pero este es el pseudocódigo referente a la generación de vecino:

```
valoración = evaluacionFactorial(pos1, pos2, solución, solución.valor)
if(valoración < solución.valor)
        Intercambio(solución, pos1, pos2)
        solución.valor = valoración
```

"Intercambio" se encarga de intercambiar en la solución las 2 posiciones ya que ahora es nuestra nueva mejor solución.

Exploración del entorno y máscara "Don´t look bits"

Utilizamos la máscara "don´t look bits" para generar los nuevos vecinos de la solución actual. Tenemos inicialmente un vector de ceros, uno por cada unidad. Si vemos que todos los vecinos posibles generados al intercambiar una unidad no mejoran a la solución actual, en el vector ponemos en su posición un 1, indicando que no se han conseguido mejora con él y que no se tenga en cuenta para generar futuros vecinos.
Para evitar que en la búsqueda "primero el mejor" le demos prioridad a los vecinos de las primeras posiciones, barajamos un vector de [0…nºunidades-1] que nos indicará en cada iteración la próxima unidad con la que probar los intercambios y generar los vecinos. Este es el pseudocódigo de la exploración de la búsqueda local utilizando la máscara de "don´t look bits":

```
aux=random[0…nºunidades]
bits=[0…0]
for(i ← 0…nºunidades && evaluaciones<max)
        pos=aux[i]
        if(bits[pos]==0)
                for(j ← 0…nºunidades && !encontrado)
                        if(j!=pos)
                                valor= evaluacionFactorial(solucion, j, pos, valor_ini)
                                evaluaciones++
                                if(valor < valor_ini)
                                        encontrado=true
                                        i=-1
                                        Intercambiar(solucion,j,pos)
                                        valor_ini = valor
                                        aux=random[0…nºunidades]
                if(¡encontrado)
                        bits[pos]=1
```

Evaluación factorizada

Hemos visto en el anterior operador que se hace una valoración factorizada al generar un vecino. Esto se hace porque teniendo ya la valoración de la solución, sería ineficiente volver a calcular el valor del vecino cuando sólo han cambiado dos posiciones. Por eso, para que resulte más eficiente, nos apoyamos en la valoración de la solución para calcular el nuevo valor en menos tiempo. Aquí tenemos el pseudocódigo:

```
a= solucion[pos1]-1
b=solucion[pos2]-1
for(i ← 0…nºunidades)
        c=solucion[i]-1
        valor += flujo[pos1][i]*(distancia[b][c]-distancia[a][c]) +
        flujo[pos2][i]*(distancia[a][c]-distancia[b][c]) +
        flujo[i][pos1]*(distancia[c][b]-distancia[c][a]) +
        flujo[i][pos2]*(distancia[c][a]-distancia[c][b])
valor += solucion.valor
```

Procedimiento y manual de la práctica:

Para desarrollar la práctica y construir los diferentes algoritmos he utilizado como base las explicaciones y pseudocódigos proporcionados en las transparencias y seminarios de la asignatura.

Manual:
Tal y como está construido el "main" de la práctica, para cada uno de los 20 casos se lanzan los 5 algoritmos (ES, BMB, ILS, ILS_ES, GRASP) 10 veces con diferentes semillas. Se calcula la media de la desviación y del tiempo para cada algoritmo en esas 10 ejecuciones.

Estos son los parámetros de construcción de los objetos relevantes:

-Evaluador(matrizFlujos,matrizDistancias)

-generadorAleatorio(semilla, mascara, prime, escala)

-Enfriamiento(nºunidades, maxEnfriamientos, maxVecinos, generadorAleatorio, evaluador, temperaturaInicial,temperaturaFinal, mejorSolucionIni, valorMejorSolucionIni)

-BMB(nºunidades, generadorAleatorio, evaluador, maxIteraciones)

-ILS(nºunidades, generadorAleatorio, evaluador, maxIteraciones)

-ILS_ES(nºunidades, generadorAleatorio, evaluador, maxEnfriamientos, maxVecinos, temperaturaInicial, temperaturaFinal)

-GRASP(nº unidades, generadorAleatorio, evaluador, alpha)

Respecto a la línea de comandos, si se quiere ejecutar sólo un caso, pasaremos como argumento el índice del caso en concreto.

La temperatura final se ha instanciado a 0.001, el parámetro del cálculo de la temperatura inicial (denominador) a 1.2039, el número de evaluaciones a 50000 y el alpha del GRASP a 0.5. Todos estos parámetros los podemos configurar o en el código o desde la línea de comando, pasándolos cómo argumentos en el orden en el que los hemos descrito.

Resultados y análisis:

Las semillas utilizadas para las 10 iteraciones han sido:
486235468, 2162468, 486363, 8135461, 35468, 22235448, 87, 666468, 48625345, 25587.

Y estos son los resultados:

Algoritmo Greedy		
Caso	**Desv**	**Tiempo**
Chr20b	365,79	0,00
Chr22a	119,91	0,00
Els19	124,41	0,00
Esc32b	90,47	0,00
Kra30b	29,61	0,00
Lipa90b	29,05	0,00
Nug25	18,53	0,00

Sko56	19,29	0,00
Sko64	17,62	0,00
Sko72	15,64	0,00
Sko100a	13,23	0,00
Sko100b	13,49	0,00
Sko100c	14,52	0,00
Sko100d	12,52	0,00
Sko100e	13,25	0,00
Tai30b	117,72	0,00
Tai50b	71,83	0,00
Tai60a	15,81	0,00
Tai256c	120,48	0,00
Tho150	17,14	0,00

Algoritmo Busqueda Local		
Caso	**Desv**	**Tiempo**
Chr20b	37,62	0,00
Chr22a	14,35	0,00
Els19	34,21	0,00
Esc32b	26,66	0,00
Kra30b	6,67	0,00
Lipa90b	22,11	0,00
Nug25	4,63	0,00
Sko56	3,31	0,00
Sko64	2,86	0,00
Sko72	2,78	0,00
Sko100a	2,14	0,01
Sko100b	2,25	0,01
Sko100c	2,49	0,01
Sko100d	2,14	0,01
Sko100e	2,65	0,01
Tai30b	16,53	0,00
Tai50b	5,43	0,00
Tai60a	5,33	0,00
Tai256c	0,95	0,17
Tho150	2,56	0,06

Algoritmo Enfriamiento		
Caso	**Desv**	**Tiempo**
Chr20b	31,88	0,00
Chr22a	8,99	0,00
Els19	35,18	0,01
Esc32b	107,14	0,00
Kra30b	3,84	0,01
Lipa90b	21,60	0,10
Nug25	12,34	0,00
Sko56	12,00	0,02
Sko64	11,06	0,03
Sko72	10,66	0,04
Sko100a	9,65	0,09
Sko100b	9,50	0,09
Sko100c	10,32	0,09
Sko100d	9,74	0,09
Sko100e	10,28	0,09
Tai30b	15,05	0,02
Tai50b	6,27	0,04
Tai60a	4,30	0,04
Tai256c	0,41	1,55
Tho150	2,06	0,29

Algoritmo BMB		
Caso	**Desv**	**Tiempo**
Chr20b	23,12	0,00
Chr22a	7,81	0,00
Els19	3,77	0,00
Esc32b	19,90	0,00
Kra30b	2,62	0,00
Lipa90b	21,78	0,20
Nug25	1,32	0,00
Sko56	1,90	0,05
Sko64	1,80	0,09
Sko72	1,58	0,13
Sko100a	1,43	0,44
Sko100b	1,43	0,44
Sko100c	1,53	0,44

Sko100d	1,54	0,43
Sko100e	1,69	0,43
Tai30b	1,24	0,01
Tai50b	2,59	0,05
Tai60a	4,41	0,05
Tai256c	0,59	4,58
Tho150	1,79	1,59

Algoritmo GRASP		
Caso	**Desv**	**Tiempo**
Chr20b	268,72	0,00
Chr22a	93,05	0,00
Els19	196,60	0,00
Esc32b	122,26	0,00
Kra30b	45,64	0,00
Lipa90b	29,68	0,00
Nug25	32,22	0,00
Sko56	24,05	0,00
Sko64	21,47	0,00
Sko72	20,13	0,00
Sko100a	18,05	0,00
Sko100b	18,67	0,00
Sko100c	18,08	0,00
Sko100d	17,81	0,00
Sko100e	19,07	0,00
Tai30b	101,30	0,00
Tai50b	78,09	0,00
Tai60a	16,59	0,00
Tai256c	106,01	0,03
Tho150	20,97	0,00

Algoritmo ILS		
Caso	**Desv**	**Tiempo**
Chr20b	18,75	0,00
Chr22a	6,05	0,00
Els19	4,35	0,00
Esc32b	8,57	0,00
Kra30b	2,19	0,00

Lipa90b	19,50	0,18
Nug25	0,90	0,00
Sko56	1,24	0,04
Sko64	1,27	0,06
Sko72	1,06	0,10
Sko100a	0,78	0,31
Sko100b	0,92	0,31
Sko100c	1,10	0,31
Sko100d	0,82	0,31
Sko100e	1,12	0,31
Tai30b	4,29	0,00
Tai50b	1,82	0,03
Tai60a	4,03	0,04
Tai256c	0,38	4,59
Tho150	0,97	1,31

Algoritmo ILS_ES		
Caso	**Desv**	**Tiempo**
Chr20b	10,65	0,13
Chr22a	4,20	0,14
Els19	19,89	0,26
Esc32b	84,28	0,17
Kra30b	2,13	0,28
Lipa90b	20,93	2,72
Nug25	8,96	0,14
Sko56	9,03	0,65
Sko64	8,44	0,88
Sko72	8,50	1,16
Sko100a	7,50	2,43
Sko100b	7,84	2,44
Sko100c	7,49	2,44
Sko100d	7,40	2,42
Sko100e	8,14	2,42
Tai30b	5,34	0,59
Tai50b	3,31	1,20
Tai60a	2,96	1,20
Tai256c	0,29	38,48
Tho150	1,09	7,35

Algoritmo	Desv	Tiempo
Greedy	62,01	0,00
BL	9,88	0,01
ES	16,61	0,13
BMB	5,19	0,44
GRASP	63,42	0,00
ILS	4,00	0,39
ILS-ES	11,41	3,37

Teniendo ya estas tablas con los resultados podemos empezar a extraer conclusiones.

La primera, como era de esperar, es que los algoritmos de Greedy y GRASP son lo que tienen un menor tiempo de ejecución. Esto se debe a su componente voraz que les permite encontrar rápidamente encontrar soluciones pues no necesitan realizar una exploración tan amplia como los demás algoritmos. En el caso del algoritmo Greedy, su "inmediato" tiempo de ejecución no compensa los malos resultados, pues observamos un 62% de desviación. No obstante, siempre es bueno tenerlo en cuenta para, por ejemplo, tomarlo como solución inicial de otros problemas.

Hablemos ahora de la búsqueda local. Valorando los algoritmos por sus resultados en desviación y tiempo, parece que la búsqueda local es el algoritmo que presenta una mejor relación entre ellos. Presenta una desviación menor del 10%, un resultado bastante aceptable, en un tiempo de ejecución medio de tan sólo 0.01 segundos. Ya vimos en la anterior práctica que para estos datos, la búsqueda local presentaba buenos resultados en muy poco tiempo.

Los resultados del enfriamiento simulado no son tan bueno cómo podríamos esperar. Hemos obtenido una desviación de 16,6% y un tiempo medio de 0.13 segundos. En el apartado sobre el esquema de enfriamiento, comentamos que habíamos tenido que modificar el esquema de enfriamiento a uno mucho más lento pues la temperatura se reducía demasiado rápido. Aun así, habiendo hecho el algoritmo más lento y permitirle una mayor exploración, este algoritmo nos ha dado un resultado mediocre.

Nuestro algoritmo BMB es realmente sencillo, se crean 25 soluciones aleatorias sobre las que aplicamos la búsqueda local y nos quedamos con la mejor. Hemos visto que la búsqueda local nos ha dado un buen resultado por lo que se espera unos buenos resultados para este algoritmo y así es. Hemos conseguido una desviación de 5,19% en un tiempo medio de 0.44 segundos.

Ya sólo nos queda hablar del ILS. Este algoritmo tiene cierta similitud con el anterior pero ahora nos movemos de una solución a otra haciendo una mutación sobre la mejor solución hasta ahora. No es una exploración tan aleatoria como en el anterior algoritmo. Viendo entonces los resultados que hemos obtenido con BMB y que ahora la exploración está más controlada, explica que obtengamos unos mejores resultados, un 4% de desviación y de media 0.39 segundos.

Sin embargo, en el ILS-ES, al estar utilizando en vez de búsqueda local, el enfriamiento resultado, perjudica considerablemente tanto los resultados como el tiempo de ejecución. Habíamos visto que el enfriamiento simulado no nos estaba dando buenos resultados, bastante peores que los de la BL, y esto hace que al llevarlo al ILS nos empeore los resultados llegando al 11,4% de desviación. A su vez, lo estamos convirtiendo en un algoritmo bastante pesado tardando de media 3.37 segundos en su ejecución.

Entonces, a modo de conclusión, nos quedaríamos con BMB e ILS pues presentan los mejores resultados y son algoritmos realmente sencillos. Diríamos de evitar el enfriamiento simulado pues no nos está dando buenos resultados, al menos para estos datos. También, si queremos priorizar el tiempo de ejecución, la búsqueda local vuelve a ser una buena opción pues tiene una desviación aceptable y la consigue en muy poco tiempo.

Para comentar un par de casos más especiales, en los datos “esc32b” con el algoritmo ES se obtiene una desviación del 107%, muy por encima de la media, lo que hace pensar que, por como son estos datos, le son realmente costosos al ES. También podemos destacar los datos “Tai256c” que han hecho que varios algoritmos se acerquen al óptimo, por debajo del 0.5% de desviación.