Práctica 3

Problema QAP:

Búsquedas por Trayectorias para
el Problema de la Asignación Cuadrática.

Óscar David López Arcos, 75571640-B
odlarcos@correo.ugr.es
Grupo A2: Martes 17:30-19:30

Índice

Descripción del problema

Este problema consiste en encontrar la asignación óptima de de n unidades a n localizaciones, conociendo la distancia entre estas últimas y el flujo existente entre las primeras. Para ello, se considerará un costo asociado a cada una de las asignaciones, que dependerá simultáneamente de la distancia y del flujo. De este modo se buscará que este costo, en función de las distancias y los flujos, sea mínimo

Es considerado como uno de los problemas de optimización más costosos (NP-completo), ya que la función de coste a minimizar es cuadrática:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} F_{ij} \cdot D_{\pi(i)\pi(j)}$$

Debemos encontrar el π (solución al problema) que genere el menor coste total posible, siendo F_{ij} el flujo que circula entre la unidad i y la j y D_{kl} la distancia existente entre la localización k y la l. Del mismo modo, el valor $\pi(i)$ representa la localización asignada a la unidad i en una determinada solución π.

Representación de soluciones

La representación interna de este problema está formada por dos matrices F y D, que representan los flujos y distancias entre cada unidad/localización (filas) y las demás (columnas). Las dimensiones de las matrices serán por tanto $n \times n$ y las diagonales principales valdrán 0. Todo esto quedará encapsulado dentro de la clase QAP junto a la implementación de los futuros algoritmos.

Cada solución π será representada por un vector, donde las posiciones identifican las unidades y el contenido las localizaciones asociadas a estas.

Función Objetivo

El coste de una solución, se calcularía de la siguiente forma:

```
S = vector de tamaño n, que representa las unidades asociadas a las localizaciones
F / D = Matriz de flujos/distancias (Explicadas en apartado anterior)

CalcularCoste(Solucion S){
        M = (Matriz n x n dimensiones inicializada a 0)
        para cada fila i de M{
                elemento de la columna S[i] = 1
        }
        Mt = Traspuesta(M)
        coste = multiplica_suma(F, M*D*Mt)
        return coste
}
```

(Función auxiliar para cálculo de Coste)

```
multiplica_suma(Matriz M1,  Matriz M2){
        if ( dimensiones M1 ≠ dimensiones M2)
                error
        resultado = 0
        para cada fila i de ambas matrices
                para cada columna j de ambas matrices
                        resultado = resultado + M1(i,j) * M2(i,j)
}
```

De todas las soluciones existentes en nuestra población, elegiremos aquella que menos coste posea.

Generación de soluciones aleatorias

Como he explicado anteriormente, una solución está representada por un vector. Como cada unidad debe estar en una localización distinta, la única forma de generar una solución aleatoria es crear un vector que contenga todas las posibles localizaciones y, posteriormente, desordenarlo aleatoriamente.

```
randomSolution ( Poblacion, tamañoPoblacion){

        v = {1,…,n}  genero solución v con los valores [1,n]
        desordeno v aleatoriamente, para conseguir una solución válida y aleatoria.
        return v
}
```

Procedimiento

La práctica ha sido desarrollada en C++, compilable por medio de un makefile (ver archivo "Leeme.txt"). Ejecutando las órdenes "make" desde el directorio Software y posteriormente "./bin/main" se repetirá el experimento analizado más adelante, con la ejecución de todos los algoritmos.

Algoritmos basados en Trayectorias usados

Enfriamiento Simulado

El esquema de enfriamiento para el algoritmo está incluido dentro del pseudocódigo de éste. Los parámetros T_0 , T_f y β se definen antes del inicio del bucle, que se repetirá mientras se cumplan las restricciones especificadas. La característica principal de este algoritmo es que no sólo se salta a una solución vecina cuando esta tiene un mejor coste, si no que podrá saltar a una peor aleatoriamente, gracias a la comparación U(0,1) ≤ $e^{-(Coste(s)-Coste(r))/T_k}$, donde U(0,1) representa un número aleatorio entre 0 y 1. La variable

T_k representa la temperatura del enfriamiento actual y, al actualizarse en cada iteración del bucle externo ($T_k = T_k/(1 + \beta * T_k)$), conforme avanza el algoritmo, es menos probable un salto a peor solución.

```
ES(){
        s = randomSolution()
        mejorSolucion = s
        maxVecinos = 10*n
        maxExitos = n
        maxIter = 50000/maxVecinos

        // Definición del esquema de enfriamiento
        T_0 = 0.3 * Coste(s)/(−log(0.3))
        T_f = 0.001
        T_k = T_0
        β = (T_0 − T_f)/(maxIter * T_0 * T_f)

        repetir{
                exitos = 0
                vecinos = 0
                repetir{
                        r  = Intercambiar dos posiciones aleatorias de s
                        si Coste(r) < Coste(s) o  U(0,1) ≤ e^−(Coste(s)−Coste(r))/T_k{
                                exitos = exitos+1
                                s = r
                                si Coste(s) < coste(mejorSolucion)
                                        mejorSolucion = s
                        }
                        vecinos = vecinos+1

                }mientras ( vecinos < maxVecinos y exitos < maxExitos)
                iteracion = iteracion+1
                T_k = T_k/(1 + β * T_k)
        }mientras (iteracion < maxIter y exitos ≠ 0)

        return mejorSolucion
}
```

Búsqueda multiarranque básica

Para la BMB he utilizado la búsqueda local implementada en la práctica 1, que explicaré a continuación. El algoritmo tiene la siguiente estructura:

```
BMB(){
        maxIter = 25
```

```
        Coste(mejorSolucion) = ∞
        para i=1 hasta i=maxIter {
                s = randomSolution()
                s = LocalSearch(s, 50000)
                si Coste(s) < Coste(mejorSolucion){
                        mejorSolucion = s
                }
        }
        return mejorSolucion
}
```

Algoritmo Búsqueda Local

Misma búsqueda local de la Práctica 1, con uso de la máscara Don't Look Bits.

```
LocalSearch(solucion, maxIter){
        DLB = vector booleano tamaño n, inicializado a FALSE
        haMejorado = TRUE
        it = 0
        mientras it < maxIter y haMejorado{
                mejorSolucion = exploracion(DLB, solucion)
                haMejorado = TRUE, si mejorSolucion ≠ solucion
                it = it+1
        }
}

exploracion(DLB, solucion){

        para cada posicion i de solucion{
                // Si la unidad no produce mejoras, acabo iteración
                si DLB_i = TRUE : acabo iteracion
                para cada posicion j de solucion{

                        si j = i: acabo iteracion
                        si DiferenciaCoste(solucion, i, j) < 0{
                                solucion = intercambiar elementos posiciones i,j de solucion
                                DLB_j = FALSE
                                return solucion
                        }
                }
                // Marco la unidad que no ha producido mejoras
                DLB_i = TRUE
        }
        return solucion}
}
```

Factorización del movimiento de intercambio

La diferencia de coste entre una solución representada por el vector v y una solución v' (calculada intercambiando los valores de las posiciones r y s de v) puede calcularse de forma mucho más eficiente con este algoritmo:

```
Diferencia_Coste(Vector v, posicion r, posicion s){

        dif = 0
        para cada componen k de v{

                if( k ≠ r  y k ≠ s)
                        dif +=  ( F[r][k] * (D[v[s]][v[k]] - D[v[r]][v[k]]) )+( F[s][k] * (D[v[r]][v[k]] - D[v[s]][v[k]]) )+
                                ( F[k][r] * (D[v[k]][v[s]] - D[v[k]][v[r]]) )+( F[k][s] * (D[v[k]][v[r]] - D[v[k]][v[s]])

        }
        return dif
 }
```

GRASP

El algoritmo GRASP es muy parecido al BMB, sólo que cada solución es generada mediante un Greedy Aleatorizado en vez de aleatoriamente. Este paso es el más complicado del algoritmo, así que lo explicaré detenidamente sirviéndome del pseudocódigo y comentarios sobre éste.

```
GRASP(){

        maxIter = 25
        Coste(mejorSolucion) = ∞

        para i=1 hasta i=maxIter {
                s = greedyGRASP()
                s = LocalSearch(s, 50000)
                si Coste(s) < Coste(mejorSolucion){
                        mejorSolucion = s
                }
        }
        return mejorSolucion
}
```

Algoritmo Greedy Aleatorizado

greedyGRASP(){

// ETAPA 1 del Algoritmo

// calculo los vectores f y d, donde cada componente corresponde a la suma de los valores de las filas de F y
// D respectivamente (válido al ser matrices simétricas):

$f_i = \sum_{j=1}^{n} F_{ij}$ con i=1…n $\qquad$ $d_k = \sum_{l=1}^{n} D_{lk}$ con k=1…n

ordeno f de mayor a menor y ordeno d de menor a mayor

// Calculo umbrales:

umbral_u = $f_1 - \alpha * (f_1 - f_n)$, $\qquad$ umbral_l = $d_1 + \alpha * (d_n - d_1)$

// Inicializo listas reducidas de candidatos

para todo f_i tal que $f_i >$ umbral_u

añadir f_i a LRCu

para todo d_i tal que $d_i <$ umbral_u

añadir d_i a LRLl

// Compruebo que el tamaño sea mínimo 2

si tamaño(LRCu) = 1

añadir f_2 a LRCu

si tamaño(LRCl) = 1

añadir d_2 a LRCl

// Escojo dos parejas aleatorias de cada lista y las añado a la solución

u1,u2 = ComponentesAleatoriasDiferentes(LRCu)

l1, l2 = ComponentesAleatoriasDiferentes(LRCl)

solucion[u1] = l1, $\qquad$ solucion[u2] = l2

// Incluyo ambas asignaciones en la lista de asignaciones realizadas

añadir (u1,l1) a la lista de asignaciones, $\qquad$ añadir (u2,l2) a la lista de asignaciones

// ETAPA 2 del Algoritmo

unidadesSinUsar = $f - \{u1, u2\}$

localizacionesSinUsar = $d - \{l1, l2\}$

mientras queden unidades sin usar{

// Crear lista de todas las posibles parejas

para todo componente i de unidadesSinUsar

para todo componente j de localizacionesSinUsar

añadir (i,j) a PosiblesParejas

```
                ordenar PosiblesParejas según costoGRASP**

                // Calcular umbral
                umbral = PosiblesParejas_1 + α * (PosiblesParejas_n − PosiblesParejas_1)

                // Definir LRC
                para todo PosiblesParejas_i tal que PosiblesParejas_i > umbral
                        añadir PosiblesParejas_i a LRCu

                ParejaAleatoria = ComponenteAleatorio de LRC

                solucion[ ParejaAleatoria.u ] = ParejaAleatoria.l

                eliminar ParejaAleatoria.u de unidadesSinUsar        eliminar ParejaAleatoria.l de localizacionesSinUsar

                añadir ( ParejaAleatoria.u , ParejaAleatoria.l ) a la lista de asignaciones
        }
        return solucion
}

**costoGRASP( Pareja (u,l), asignacionesRealizadas) {
        CostoPareja = 0
        para cada componente a de asignacionesRealizadas{
                CostoPareja = CostoPareja + F_{u,a.u} * D_{l,a.l}
        }
        return CostoPareja
}
```

ILS

Tanto el algoritmo ILS como el ILS-ES tienen la peculiaridad de no partir en cada iteración de nuevas soluciones aleatorias, si no de aplicar una mutación fuerte a la actual. En este caso la mutación fuerte consistirá en desordenar un subconjunto del vector solución de tamaño n/4. Para ILS-ES en vez de aplicar una BL, se aplica sobre la solución el algoritmo ES.

```
ILS(){
        it = 0, itMax = 24
        mejorSolucion = randomSolution()
        solucion = LocalSearch(mejorSolucion, 50000)

        mientras( it < itMax){

                si Coste(solucion) < Coste(mejorSolucion)
                        mejorSolucion = solucion
```

```
            solucion = mejorSolucion
            solucion = mutacionILS(solucion)
            solucion = LocalSearch(solucion, 50000)

            it = it+1
        }
        return solucion
}
```

Mutacion fuerte empleada en ILS

```
mutacionILS(solucion){

        random = numero aleatorio entre 1 y n/4
        desordenar aleatoriamente las posiciones de solucion entre random y (random+n/4)
        return solucion
}
```

Experimento y análisis

Los resultados obtenidos para los diferentes conjuntos de datos y algoritmos han sido:

	ES			BMB			GRASP		
Archivo	**Coste**	**Desv.**	**Tiem.**	**Coste**	**Desv.**	**Tiem.**	**Coste**	**Desv.**	**Tiem.**
Chr22a	8518	38,37	0,01	6720	9,16	0,01	6520	5,91	0,02
Chr22b	8594	38,75	0,01	6660	7,52	0,01	6584	6,30	0,02
Chr25a	11632	206,43	0,01	5436	43,20	0,02	4954	30,51	0,04
Esc128	240	275,00	0,03	64	0,00	1,88	64	0,00	6,45
Had20	7288	5,29	0,00	6926	0,06	0,01	6930	0,12	0,02
Lipa60b	3209634	27,36	0,01	3014219	19,61	0,29	3011999	19,52	0,65
Lipa80b	10007362	28,90	0,01	9435181	21,53	0,70	9437618	21,56	1,72
Nug28	6262	21,22	0,01	5302	2,63	0,03	5214	0,93	0,06
Sko81	104902	15,28	0,01	92606	1,77	1,44	92804	1,98	2,39
Sko90	132970	15,09	0,02	117608	1,80	2,12	117430	1,64	3,55
Sko100a	174604	14,87	0,02	154644	1,74	3,15	154376	1,56	5,31
Sko100f	170964	14,71	0,02	151556	1,69	2,96	151454	1,62	5,11
Tai100a	23645944	12,32	0,02	21776052	3,44	1,36	21729398	3,22	3,65
Tai100b	1637410748	38,06	0,02	1205975014	1,68	2,83	1207087789	1,78	5,23

	ES			BMB			GRASP		
Tai150b	629923101	26,26	0,03	507937134	1,81	12,88	510580514	2,34	22,54
Tai256c	51306166	14,63	0,10	44932974	0,39	23,87	44925182	0,37	84,64
Tho40	304100	26,44	0,01	246004	2,28	0,12	249178	3,60	0,21
Tho150	9592572	17,94	0,03	8270858	1,69	13,28	8308280	2,15	22,75
Wil50	53412	9,41	0,01	49324	1,04	0,26	49298	0,99	0,45
Wil100	295360	8,18	0,02	275668	0,96	3,09	275658	0,96	5,24

	ILS			ILS-ES		
Archivo	**Coste**	**Desviación**	**Tiempo**	**Coste**	**Desviación**	**Tiempo**
Chr22a	6432	4,48	0,01	9074	47,40	0,13
Chr22b	6780	9,46	0,01	8566	38,30	0,13
Chr25a	4012	5,69	0,01	10610	179,50	0,14
Esc128	74	15,63	1,69	252	293,75	0,62
Had20	6922	0,00	0,01	7240	4,59	0,10
Lipa60b	3005180	19,25	0,21	3208734	27,32	0,24
Lipa80b	9434530	21,52	0,54	9947471	28,12	0,33
Nug28	5298	2,56	0,02	6146	18,97	0,13
Sko81	92236	1,36	0,69	104908	15,29	0,34
Sko90	116996	1,27	0,96	130360	12,83	0,38
Sko100a	154156	1,42	1,42	173844	14,37	0,44
Sko100f	150772	1,16	1,45	169142	13,49	0,44
Tai100a	21768248	3,40	1,05	23644650	12,31	0,44
Tai100b	1209799198	2,01	1,51	1637409074	38,06	0,44
Tai150b	504950561	1,21	6,88	620665712	24,41	0,73
Tai256c	44926022	0,37	16,76	49892248	11,47	2,58
Tho40	244374	1,60	0,07	298978	24,31	0,19
Tho150	8302250	2,08	6,76	9589048	17,90	0,74
Wil50	49330	1,05	0,14	53790	10,19	0,21
Wil100	274546	0,55	1,49	293206	7,39	0,43

Haciendo la media de la desviación y tiempo, obtenemos la siguiente tabla:

Algoritmo	Desviación media	Tiempo medio
ES	**42,72**	**0,02**
BMB	**6,20**	**3,52**
GRASP	**5,35**	**8,50**
ILS	**4,80**	**2,08**
ILS-ES	**42,00**	**0,46**

Las metaheurísticas basadas en trayectorias funcionan muy bien para el problema QAP. El algoritmo del Enfriamiento Simulado, siendo una implementación muy básica, es el más rápido y genera resultados similares a los algoritmos genéticos de la práctica anterior. Los siguientes algoritmos que utilizan una mejor búsqueda local, son capaces de explotar el espacio de búsqueda y encontrar solución más cercana al óptimo.

En general, el equilibrio entre explotación y exploración está muy presente en estos algoritmos, y se consigue generando muchas soluciones que estén en diferentes zonas del espacio (exploración) y aplicando una buena búsqueda local para obtener el óptimo local de este espacio (explotación).

Aunque todos ofrecen una relación Coste/Tiempo muy buena, el Algoritmo ILS posee la menor desviación media y es superado en tiempo únicamente por los algoritmos que no usan la BL, cuyo coste es mucho mayor. Podemos deducir entonces que la mutación fuerte aplicada en este algoritmo combinada con una buena búsqueda local es la causante de los buenos resultados. La mutación provoca exploración pero, al no partir de soluciones completamente nuevas en cada iteración, se mantiene información de buenas asignaciones provocadas por anteriores explotaciones. De este modo se consigue vencer a BMB o incluso a GRASP, a pesar del partir de soluciones Greedy.

Como conclusión podríamos decir que, de todos los algoritmos trabajados en la asignatura, estos serían los ideales para utilizar en el problema de Asignación Cuadrática.