PRÁCTICA 3

ES, BMB, GRASP, ILS, ILSES para QAP – 3º Grupo 1

Julio A. Fresneda García – 49215154F

DESCRIPCIÓN DEL PROBLEMA

El problema de Asignación Cuadrática (QAP) es uno de los problemas de optimización combinatoria más complejos. Es NP-completo.

En el QAP, lo que tratamos de hacer es asignar n unidades a una cantidad n de localicaziones, donde se considera un coste aosicado a cada una de las asignaciones. Este coste dependerá de la distancia entre localizaciones y del flujo en-tre unidades.

Lo que se busca es que el coste total, en función de la distancia y el flujo, sea mínimo.

En este problema contamos con dos matrices, la matriz de flujos y la matriz de distancias, que nos sirven para obtener el flujo entre unidades y la distancia entre localizaciones.

DESCRIPCIÓN DE LA APLICACIÓN DE LOS ALGORITMOS AL PROBLEMA

La función objetivo es la siguiente:

$$QAP = \min_{S \in \Pi_N} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} \cdot d_{S(i)S(j)} \right)$$

donde f es la matriz de flujos, d la matriz de distancias y S el vector solución.

Vamos a considerar lo siguiente:

Tenemos dos matrices, una para representar los flujos entre unidades, y otra matriz para representar la distancia entre localizaciones.

La distancia entre las localizaciones i y j se comprueba en la matriz de distancias, en la posición [i][j]. De igual forma, el flujo entre las unidades k y l se comprueba en la matriz de flujos, en la posición [k][l].

Las bases de datos que usaremos para comprobar nuestros algoritmos son archivos .dat que tienen el siguiente formato:

Encabeza un número que representa el tamaño del problema. Le sigue una línea en blanco, y después la matriz de flujos. Le sigue una línea en blanco (en algunos archivos), y la matriz de distancias.

Para representar las posibles soluciones, usaremos un vector de enteros, en el cual las posiciones del vector representan las unidades, y el contenido de cada posición representa a la localización asignada a esa unidad. Por ejemplo, si tenemos como solución {2,3,1,0} significa que la unidad 0 está asociada a la localización 2; la unidad 1, a la localización 3; etc.

Para calcular el coste de una solución en concreto, usamos la siguiente fórmula:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij} \cdot d_{S(i)S(j)}$$

En pseudocódigo, el cálculo de coste de una solución sería así:

```
Coste ← 0
for i=0 to n
        for j=0 to n
                if i != j
                        coste = coste + flujos[i][j]*distancias[solución[i]][solución[j]]
                endif
        endfor
                                                endfor
```

Para llegar a la solución S mínima, vamos a implementar los algoritmos que explicaremos en los siguientes apartados.

Para representar las posibles soluciones, usaremos un vector de enteros, en el cual las posiciones del vector representan las unidades, y el contenido de cada posición representa a la localización asignada a esa unidad. Por ejemplo, si tenemos como solución {2,3,1,0} significa que la unidad 0 está asociada a la localización 2; la unidad 1, a la localización 3; etc.

Antes de aplicar los algoritmos BMB e ILS, partimos de unas primeras soluciones. Estas primeras soluciones se crean automáticamente cuando creamos un objeto QAP en nuestro código, el cual genera una primera asignación de localizaciones en unidades aleatoria. La generación es la siguiente.

```
for i=0 to n
    ordenados[i] ← i
endfor

while ordenados.Length != 0
    i ←random(0,ordenados.Length)
    solucion ←ordenados[i]
    remove ordenados[i]
endwhile

return solucion
```

DESCRIPCIÓN DE LOS ALGORITMOS EN PSEUDOCÓDIGO.

Enfriamiento Simulado.

En este apartado detallaré las partes más importantes del algoritmo de Enfriamiento Simulado, en pseudocódigo.

Cálculo de temperatura inicial.

El cálculo es una operación sencilla. En nuestro caso tanto fi como eta toman valores por defecto de 0,3.

```
CalculoTo( fi, eta )
{
        return ( eta * CosteSolución ) / -ln( fi )
}
```

Intercambio de posiciones para encontrar un vecino.

En éste método simplemente intercambiamos dos posiciones del vector de asignaciones de localizaciones a unidades.

```
Neighborhood_Op( List solucion, costeMovimiento )
{
        i <- random(0, length(solución) )
        j <- random(0, length(solución) )
        while( i == j ) j <- random(0, length(solución) )

        temp <- solución[i]
        solución[i] <- solución[j]
        solución[j] <- temp

        coste = CosteMovimiento( i, j )
        return solución
}
```

Esquema general de enfriamiento.

Este es el esquema general del algoritmo de ES. Lo primero que se hace es calcular la temperatura inicial a partir de la fórmula dada, normalmente usamos fi y eta como 0.3. La temperatura final será de 0.001, y el número de enfriamientos de 50000 partido por el cuadrado del tamaño del problema.

Empezamos generando una solución inicial aleatoria, y hacemos un bucle do-while hasta que la temperatura sea menor que la temperatura final.

Dentro del bucle do-while, hacemos otro bucle: Desde 0 hasta el número de enfriamientos. En ese bucle, obtenemos una solución vecina (intercambiando dos asignaciones de la solución actual), y si esta solución vecina es mejor que la anterior, la sustituimos. La sustitución también está sujeta a cierta probabilidad, en este caso se sustituye si un número aleatorio entre 0 y 1 es menor que exp(-coste_movimiento/T) (dejamos k=1). Si este nuevo vecino es mejor que la mejor solución encontrada hasta ahora, se reemplaza.

Cuando acaba el for, bajamos la temperatura. Con el sistema de Cauchy la temperatura bajaba demasiado rápido, por lo que aquí en cada bucle do-while se multiplicará por 0,9.

Al acabar el do-while, devolvemos la mejor solución encontrada.

```
ResolverES
{
        To <- CalcularTo( fi, eta )
        T <- To
        Tf <- 0.001
        numEnfriamientos <- 50000 / (tamProblema)^2
        s <- soluciónActual
        bestSolution <- s

        Do
        {
                For( i=0 to numEnfriamientos )
                {
                        s' <- neighborhood_Op( s, coste_movimiento )
                        rand <- random( 0, 1 )
                        if( coste_movimiento < 0 or rand < exp( -coste_movimiento / (k*T) )
                        {
                                s <- s'
                                if( Coste( s ) < Coste( bestSolution )
                                {
                                        bestSolution <- s
                                }
                        }
                }
                T <- T*0.9
        } while( T > Tf )
}
```

Búsqueda Multiarranque Básica

Una BMB es simplemente ejecutar un número determinado de veces (en nuestro caso 25) Búsqueda Local, y quedarnos con la mejor solución.

Recordemos que el método de búsqueda local es el siguiente

```
hayMejora ←true

for i=0 to n
    dlb[i] ←false
endfor

while hayMejora
    hayMejora ←false
    for i=0 to n
        if dlb[i] = false
            improveFlag ←false
            for j=0 to n
                if i≠j
                    if CosteIntercambio(i,j) < 0
                        intercambiar(i,j)
                        dlb[i] ←false
                        dlb[j] ←false
                        improveFlag ←true
                        hayMejora ←true
                    endif
                endif
            endfor
        endif
    endfor
endwhile
```

Y el método que calcula el coste después de un intercambio, de forma factorizada, para ahorrarnos tiempo de computación, es el siguiente.

```
    costeIntercambio r, s
        coste ←0
        for k=0 to n
            if k ≠r and k ≠s
                coste←coste + flujos[r][k]*(distancias[solucion[s]][solucion[k]] -
distancias[solucion[r]][solucion[k]])
                coste←coste + flujos[s][k]*(distancias[solucion[r]][solucion[k]] -
distancias[solucion[s]][solucion[k]])
                coste←coste + flujos[k][r]*(distancias[solucion[k]][solucion[s]] -
distancias[solucion[k]][solucion[r]])
                coste←coste + flujos[k][s]*(distancias[solucion[k]][solucion[r]] -
distancias[solucion[k]][solucion[s]])
            endif
        endfor

        return coste
    end
```

GRASP

En resumen, GRASP consiste en obtener con greedy una solución aleatoria pero de entre las mejores (usando los potenciales) y aplicarle búsqueda local. Este proceso se hace reiteradamente hasta obtener una solución final.

Voy a explicarlo más detalladamente con ayuda de pseudocódigo.

Esquema general:

El proceso lo repetiremos 25 veces en este caso. Primero, resolvemos los potenciales, los cuales nos ayudarán a elegir con Greedy en las etapas.

Para guardar estos potenciales, guardaremos en dos listas de candidatos LCu y LCl estos potenciales junto a su índice como pareja, así podremos borrar elementos de las listas de candidatos sin que perdamos información de sus índices.

Después, llevamos a cabo las etapas 1 y 2, y optimizamos la solución obtenida con búsqueda local.

Repetimos este proceso 25 veces, y nos quedamos con la mejor solución.

```
GRASP()
{
        for( x=0 to numIteraciones )
        {
                ResolverPotenciales( potencialFlujos, potencialDistancias )
                for ( i=0 to tamProblema )
                        LCu.Add ( pair i, potencialFlujos[i] )
                        LCl.Add( pair i, potencialDistancias[i] )
                }
                Etapa1()
                S <- Etapa2()
                Solucion <- S
                OptimizacionBL( Solucion )
                If( Coste(Solucion) < Coste(BestSolucion) ) BestSolucion <- Solucion
        }
        return BestSolucion
}
```

Ahora vamos a explicar la primera fase de GRASP: La etapa 1.

En la etapa 1 lo que buscamos es empezar a completar una solución parcial con dos asignaciones. Para elegir estas asignaciones, cogemos de entre todas las posibles asignaciones aquellas cuya unidad esté por encima de cierto umbral, y cuya localización esté por debajo de cierto umbral. Si sólo hay 1 localización o 1 unidad que cumplan estas condiciones, la otra la asignación la hacemos completamente aleatoria.

Primero, lo que se hace es obtener los umbrales, los cuales están relacionados con los potenciales máximos y mínimos. Después se forma la lista restringida de candidatos, formada por las unidades y localizaciones que hayan sobrepasado el umbral. Para formarla necesitaremos las listas de candidatos que obtuvimos antes de la etapa 1.

Una vez tenemos esta lista, cogemos dos asignaciones de forma aleatoria de entre la lista y la añadimos a la solución parcial.

Ya tenemos el principio de nuestra solución.

```
Etapa1()
{
        CalcularMinimosYMaximos()
        Umbral_U <- máximo_LRCu – 0.3*(máximo_LRCu-mínimo_LRCu)
        Umbral_L <- minimo_LRCl + 0.3*(máximo_LRCl-mínimo_LRCl)

        for( i=0 to length(LCu))
        {
                If( LCu[i] >= Umbral_U )   LRCu.Add( LCu[i] )
                If( LCl[i] >= Umbral_L )   LRCl.Add( LCl[i] )
        }
        SoluciónParcial <- { Random(LRCu), Random(LRCl) } // Estos elementos se borran de LCRu y LCRl

        If( length( LRCu ) > 1 and length( LRCl ) > 1 ) SoluciónParcial <- { Random(LRCu), Random(LRCl) }
        else SoluciónParcial <- { Random(LCu), Random(LCl) }


}
```

En la etapa 2, terminaremos nuestra solución parcial.

Empezamos creando dos listas, notu y notl. Estas listas contendrán las unidades y localizaciones ya asignadas, por lo que no las podremos usar.

Continuamos creando la lista de candidatos LC, la cual contiene todas las posibles combinaciones de unidades y localizaciones (siempre y cuando no estén en notu y notl ni en la solución parcial).

También crearemos una lista de costes, en la cual cada coste se calcula con la siguiente fórmula.

$$C_{ik} = \sum_{(j,l)\in S} f_{ij} \cdot d_{kl}$$

Esta lista de costes nos dirá qué asignaciones son las mejores.

Una vez habiendo obtenido la lista de candidatos y los costes, crearemos una lista restringida de candidatos, la cual contendrá los elementos de la lista de candidatos cuyo coste sea mayor a cierto umbral.

De entre todos los elementos de esta lista restringida, elegimos uno al azar y lo adherimos a la solución. Nuestra solución parcial ahora tiene una asignación más.

Cuando hayamos completado la solución parcial, tendremos una solución completa, la cual el método devolverá.

```
Etapa2()
{
        notu <- new List
        notl <- new List
        while( length(SolucionParcial) != tamProblema )
        {
                for( i=0 to tamProblema ){
                        for( j=0 to tamProblema ){
                                if( !notu.Contains( i ) and ¡notl.Contains( j ) ) LC.Add( asignación( i, j ) )
                        }
                }

                for( i=0 to length( LC ) )
                {
                        Costes.Add( Coste(i) )
                }

                for( i=0 to length( LC ) )
                {
                        If( Costes[i] <= costeminimo + 0.3*( costemaximo – costeminimo ) ) LRC.Add( LC[i] )
                }

                SolucionParcial.Add( Random LRC[i] )

                notu.Add( unidad de LRC[i] )
                notl.Add( localización de LRC[i] )

        }
        return SoluciónParcial
}
```

ILS

El algoritmo ILS se puede resumir en generar una solución aleatoria, y en bucle aplicarle una mutación y búsqueda local. En cada iteración guardamos siempre el mejor resultado, el cual usaremos para la siguiente iteración. En este caso realizaremos 25 iteraciones. Vamos a verlo en pseudocódigo.

```
ILS()
{
        qap <- soluciónAleatoria
        bestSolution <- qap

        for( i=0 to numIteraciones )
        {
                Mutar(qap)
                qap <- BusquedaLocal(qap)
                if( Coste(qap) < Coste(bestSolution) ) bestSolution <- qap
                else qap  <- bestSolution
        }
        return bestSolution
}
```

Vamos a ver ahora nuestro operador de mutación. La mutación consiste simplemente en coger una subcadena de las asignaciones de tamaño n/4 y mezclarla. De esta forma tenemos una mutación bastante potente.

```
Mutar(  )
{
        tamSubcadena <- length(solución) / 4
        inicioSubcadena <- random(0, length(solución) – tamSubcadena + 1)
        fin <- inicioSubcadena + tamSubcadena

        for( i=inicio to fin )
        {
                subcadena.Add( solución[i] )
        }

        while( length( subcadena ) ¡= 0 )
        {
                Index <- random( 0, length( subcadena ) )
                Solución[ inicioSubcadena + length(subcadena) -1] = subcadena[index]
                subcadena.RemoveAt( index )
        }

        return solucion
}
```

ILS-ES

ILS-ES es una variante de ILS donde en vez de usar Búsqueda Local, usamos Enfriamiento Simulado.

DESARROLLO DE LA PRÁCTICA

El desarrollo de esta práctica se ha llevado a cabo en C#. Hay varios archivos fuente: El lector de los archivos .dat; el código del QAP, que contiene las matrices y la solución, así como los algoritmos para encontrar soluciones: ES, BMB, etc.

CÁLCULO DE LA BONDAD DE UN ALGORITMO

Para calcular la bondad de un algoritmo, vamos a usar la media de desviaciones típicas de los costes obtenidos por este algoritmo respecto los mejores costes conocidos. Cuanto menor sea la desviación típica, mejor es el algoritmo.

Para calcular cada coste, usaremos una base de datos con archivos de distintos tamaños.

La media de desviaciones típicas se calcula de la siguiente forma.

```
desviacion ←0
for i=0 to n
   desviacion ←desviacion + 100*((costes[i]-mejoresCostes[i])/mejoresCostes[i])
endfor

desviacion ←desviacion/n
```

Donde costes es un vector que contiene los costes obtenidos por el algoritmo, y mejoresCostes es el vector que contiene los mejores costes conocidos.

Las medidas realizadas están en las siguientes hojas del documento.

En resumen, hemos obtenido muy buenos resultados y muy parecidos en general. Enfriamiento simulado y BMB han sido los algoritmos que (en comparación) han obtenido peores resultados. ILS-ES ha sido el mejor algoritmo con diferencia, tiene una ventaja de más de 2 puntos de desviación respecto al resto. A todos los algoritmos se les ha resistido mucho la base de datos “Chr25a”, para la cual todos excepto ILS-ES han tenido una desviación de 40 puntos aproximadamente. Aquí ILS-ES sólo ha obtenido 7 puntos, lo cual ha hecho que la media de desviación sea mucho menor que la del resto.

El algoritmo que más ha tardado es GRASP. Esto posiblemente sea porque en la implementación se necesita combinar todas las posibles asignaciones de localizaciones en unidades para obtener la lista de candidatos de la etapa 2, lo cual en bases de datos grandes es bastante costoso.

En conclusión, si tuviera que elegir el mejor algoritmo elegiría ILS-ES, pero cada problema es un mundo, y la mejor decisión es usar varios y quedarnos con el mejor resultado.

Algoritmo ES.

Media Desv:	**5,96**
Media Tiempo:	**0,00**

ES			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr22a	6370	3,48	00:00:01.9361033
Chr22b	6406	3,42	00:00:01.2528441
Chr25a	5372	41,52	00:00:00.9787046
Esc128	68	6,25	00:00:00.7661936
Had20	6948	0,38	00:00:00.9351706
Lipa60b	3017611	19,74	00:00:01.3288129
Lipa80b	9418505	21,31	00:00:01.4448926
Nug28	5218	1,01	00:00:00.9244523
Sko81	92586	1,75	00:00:01.1402336
Sko90	116672	0,98	00:00:01.1868845
Sko100a	153630	1,07	00:00:01.1624126
Sko100f	150948	1,28	00:00:01.1600182
Tai100a	21802556	3,56	00:00:01.4755597
Tai100b	1232287006	3,90	00:00:01.7913900
Tai150b	515999372	3,43	00:00:02.0341678
Tai256c	45226832	1,04	00:00:01.8068259
Tho40	246236	2,38	00:00:01.2577610
Tho150	8251928	1,46	00:00:01.5221491
Wil50	49248	0,88	00:00:01.1672228
Wil100	273924	0,32	00:00:01.2109890

Algoritmo BMB

BMB			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr22a	6798	10,43	00:00:00.0771173
Chr22b	6412	3,52	00:00:00.0640746
Chr25a	5402	42,31	00:00:00.1093933
Esc128	64	0,00	00:00:09.5059673
Had20	6922	0,00	00:00:00.0607803
Lipa60b	3014744	19,63	00:00:01.3423041
Lipa80b	9429364	21,45	00:00:03.3131781
Nug28	5210	0,85	00:00:00.1484679
Sko81	92296	1,43	00:00:05.2728936
Sko90	117504	1,71	00:00:07.2315500
Sko100a	154816	1,85	00:00:10.4437649
Sko100f	151552	1,69	00:00:10.6650678
Tai100a	21765034	3,38	00:00:06.2352633
Tai100b	1211052987	2,11	00:00:11.2194451
Tai150b	509714327	2,17	00:00:32.9914026
Tai256c	44927060	0,37	00:01:06.8899755
Tho40	247468	2,89	00:00:00.5055726
Tho150	8251732	1,45	00:00:32.3570343
Wil50	49094	0,57	00:00:01.0563382
Wil100	275398	0,86	00:00:10.3926319
Media Desv:	**5,93**		
Media Tiempo:	**0,00**		

Algoritmo GRASP

Media Desv:	**5,79**
Media Tiempo:	**0,00**

GRASP			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr22a	6594	7,12	00:00:00.3174358
Chr22b	6742	8,85	00:00:00.2716132
Chr25a	5284	39,20	00:00:00.3334354
Esc128	64	0,00	00:00:42.7087959
Had20	6924	0,03	00:00:00.2601422
Lipa60b	3003398	19,18	00:00:03.2615763
Lipa80b	9399205	21,06	00:00:08.6629029
Nug28	5298	2,56	00:00:00.4449847
Sko81	92382	1,52	00:00:11.0374139
Sko90	117266	1,50	00:00:15.8872875
Sko100a	154598	1,71	00:00:23.0899608
Sko100f	150828	1,20	00:00:23.4758263
Tai100a	21751570	3,32	00:00:19.3093872
Tai100b	1206555188	1,73	00:00:24.1215670
Tai150b	510336134	2,29	00:01:35.8783352
Tai256c	44908170	0,33	00:09:14.2782512
Tho40	241902	0,58	00:00:01.0389375
Tho150	8285584	1,87	00:01:31.0315741
Wil50	49258	0,91	00:00:02.0558314
Wil100	275278	0,82	00:00:22.7602365

Algoritmo ILS

Media Desv:	**5,14**
Media Tiempo:	**0,00**

ILS			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr22a	6552	6,43	00:00:00.0519304
Chr22b	6664	7,59	00:00:00.0487432
Chr25a	5162	35,99	00:00:00.0835907
Esc128	64	0,00	00:00:08.4067047
Had20	6924	0,03	00:00:00.0348940
Lipa60b	3005457	19,26	00:00:01.1130439
Lipa80b	9383739	20,86	00:00:02.7076239
Nug28	5218	1,01	00:00:00.1064767
Sko81	91894	0,98	00:00:03.3624880
Sko90	116634	0,95	00:00:04.6576105
Sko100a	152926	0,61	00:00:06.3309676
Sko100f	150008	0,65	00:00:06.7576484
Tai100a	21655524	2,86	00:00:05.2463387
Tai100b	1212013190	2,19	00:00:07.1829536
Tai150b	503169973	0,86	00:00:26.2325203
Tai256c	44935740	0,39	00:01:03.7074293
Tho40	241214	0,29	00:00:00.3422196
Tho150	8221740	1,09	00:00:24.7544629
Wil50	49008	0,39	00:00:00.7386801
Wil100	274148	0,41	00:00:06.3781829

Algoritmo ILS-ES

Media Desv:	**3,16**
Media Tiempo:	**0,00**

ILS-ES			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr22a	6202	0,75	00:00:27.6727964
Chr22b	6424	3,71	00:00:27.7063755
Chr25a	4088	7,69	00:00:23.4561131
Esc128	64	0,00	00:00:18.0087982
Had20	6922	0,00	00:00:23.0334783
Lipa60b	2992447	18,74	00:00:33.8881050
Lipa80b	9363290	20,60	00:00:37.1368207
Nug28	5194	0,54	00:00:23.5534698
Sko81	91628	0,69	00:00:28.7535490
Sko90	116512	0,85	00:00:30.9832955
Sko100a	152916	0,60	00:00:30.8454136
Sko100f	150384	0,90	00:00:29.6958539
Tai100a	21672000	2,94	00:00:38.3398789
Tai100b	1194772456	0,74	00:00:43.8677944
Tai150b	506963282	1,62	00:00:44.1522025
Tai256c	44982884	0,50	00:00:42.8449955
Tho40	241866	0,56	00:00:29.1849837
Tho150	8244620	1,37	00:00:37.1599772
Wil50	48872	0,11	00:00:29.0460978
Wil100	274022	0,36	00:00:29.8716562