PRÁCTICA 1

PROBLEMA DE ASIGNACIÓN CUADRÁTICA (QAP)

JULIO A. FRESNEDA GARCÍA – 49215154F – JULIOFRESNEDAG@CORREO.UGR.ES – ALGORITMOS GREEDY Y BÚSQUEDA LOCAL – GRUPO 1 – LUNES 5:30-7:30

ÍNDICE

C) 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 entre 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

D) 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.

Para llegar a la solución S mínima, vamos a implementar dos algoritmos: Greedy y Búsqueda Local. Para implementar los algoritmos 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.

Antes de aplicar los algoritmos, usamos una primera solución. Esta solución es aleatoria, y se genera de la siguiente manera:

```
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
```

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[solucion[i]][solucion[j]]
        endif
    endfor
endfor
return coste
```

Para encontrar la solución con menor coste posible, vamos a usar dos algoritmos: Greedy y Búsqueda Local. Ambos algoritmos son heurísticos, por lo que no tienen por qué encontrar la solución óptima.

GREEDY

En términos generales, el algoritmo greedy es una estrategia de búsqueda por la cual se sigue una heurística que consiste en elegir la opción óptima en cada paso localmente, sin tener en cuenta los pasos pasados o futuros.

En nuestro caso, el agoritmo greedy consiste en empezar a fabricar una solución, cogiendo la mejor asignación unidad-localización en cada paso. Es decir, iterativamente, asociamos la unidad con mayor flujo con el resto de unidades, con la localización con menor distancia con el resto de localizaciones.

Para saber que unidad tiene más flujo con el resto, y qué localización tiene menor distancia con el resto, debemos calcular el potencial de flujo y de distancia.

El potencial de flujo de una unidad se calcula con la sumatoria de flujos del resto de unidades con esta unidad.

El potencial de distancia de una localización se calcula con la sumatoria de distancias del resto de localizaciones con esta localización.

Cuanto mayor potencial de flujo tiene una unidad, más importante es la unidad en el intercambio de flujos, y cuanto menor potencial de distancias tiene una localización, menos distancia hay entre esta localización y el resto.

Por lo tanto, el algoritmo Greedy, iterativamente, asociará la unidad con mayor potencial de flujo con la localización con menor potencial de distancia, hasta asociar todas las unidades con todas las localizaciones.

BÚSQUEDA LOCAL

Para explicar el algoritmo de Búsqueda Local hay que definir lo que es el entorno de una solución.

El entorno de una solución S está formado por las soluciones accesibles desde ella a través de un movimiento de intercambio.

Tenemos S = [S(0), S(1), S(2), …, S(n-1)]

Un movimiento de intercambio consiste en intercambiar de posición dos localizaciones, es decir, asignar una unidad la localización de otra unidad y viceversa.

Si intercambiamos las unidades 1 y 2, la nueva solución sería S = [S(0), S(2), S(1), …, S(n-1)]. Ésta nueva solución cumple las condiciones del problema, si la solución original las cumple.

Sabiendo esto, vamos a describir nuestra Búsqueda Local.

En nuestro algoritmo de búsqueda local, en cuanto se genera una solución vecina mejor que la actual (con menor coste), se aplica el movimiento y se pasa a la siguiente iteración. La búsqueda se detiene cuando se ha explorado el entorno completo sin encontrar mejora.

Para, a partir de una solución, calcular el coste de la nueva solución vecina, no calculamos el coste desde 0, usamos una factorización. Esta operación se describirá en pseudocódigo en el siguiente apartado.

Este algoritmo consiste en tres bucles: El bucle externo y dos bucles internos.

Los bucles internos sirven para, para cada unidad, ver si hay posibles intercambios en los que el coste total baje, y si los hay, intercambiarlos. El bucle externo repite este proceso hasta que no haya ninguna mejora.

Se usa la técnica “Don’t Look Bits”, que, en nuestro caso, consiste en marcar cada unidad que no mejore con ningún posible intercambio, para no volver a tenerla en cuenta en las siguientes iteraciones del bucle externo. Si esta unidad marcada participa en un intercambio para disminuir el coste, se vuelve a desmarcar.

E) PSEUDOCÓDIGO DE LOS ALGORITMOS

GREEDY

El algoritmo Greedy funciona así:

```
calcularPotenciales()

indexMaxPF = 0
indexMinPD = 0

for i=0 to n
    maxPotencialFlujo = -1
    minPotencialDistancia = -1

    for j=0 to n
        if potencialFlujo[j] > maxPotencialFlujo
            maxPotencialFlujo = potencialFlujo[j]
            indexMaxPF = j
        endif
    endfor

    for j=0 to n
        if potencialDistancia[j] > -1
            if potencialDistancia[j] < minPotencialDistancia or minPotencialDistancia == -1

                minPotencialDistancia = potencialDistancia[j]
                indexMinPD = j
            endif
        endif
    endfor

    potencialFlujo[indexMaxPF] = -1
    potencialDistancia[indexMinPD] = -1

    solucion[indexMaxPF] = indexMinPD

endfor
```

El método calcularPotenciales() para calcular los potenciales, es el siguiente.

```
for i=0 to n
    potencialFlujo[i] ←0
    potencialDistancia[i] ←0
endfor

for i=0 to n
    for j=0 to n
        potencialFlujo[i] = potencialFlujo[i] + flujos[i][j]
        potencialDistancia[i] = potencialDistancia[i] + distancias[i][j]
    endfor
endfor
```

BÚSQUEDA LOCAL

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
```

El método para intercambiar las posiciones i y j es el siguiente

```
Intercambiar i, j
    temp ←solucion[i]
    solucion[i] ←solucion[j]
    solucion[j] ←temp
end
```

El método que calcula el coste después de un intercambio, de forma factorizada, 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
```

F) ALGORITMO DE COMPARACIÓN

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.

G) DESARROLLO DE LA PRÁCTICA

El desarrollo de esta práctica se ha llevado a cabo en C#. Hay 4 archivos fuente: Lector.cs, lector de los archivos .dat; QAP.cs, que contiene las matrices y la solución, así como los algoritmos para encontrar soluciones; MedirDatos.cs, el medidor de tiempos, que ejecuta los algoritmos para cada uno de los archivos .dat; y Program.cs, el código main, que ejecuta el medidor de tiempos.

No se han usado frameworks ni código externo.

Para replicar el proceso, hay que cambiar las rutas de archivos .dat del archivo MedirTiempos.cs por las rutas donde tengamos los archivos .dat, y compilar y ejecutar el archivo Program.cs.

H) EXPERIMENTOS Y ANÁLISIS DE RESULTADOS

Los resultados de aplicar los algoritmos son los siguientes.

Número de casos: 20

Greedy:	Caso:	Desviación	Coste	Mejor Coste	Tiempo (s)
	chr22a.dat	119.9155	13538	6156	0.0004026
	chr22b.dat	92.79948	11942	6194	0.0000247
	chr25a.dat	362.4868	17556	3796	0.000034
	esc128.dat	121.875	142	64	0.0006156
	had20.dat	10.11268	7622	6922	0.000021
	lipa60b.dat	28.24952	3232061	2520135	0.0001547
	lipa80b.dat	29.41676	10047868	7763962	0.0002668
	nug28.dat	19.08633	6152	5166	0.0000366
	sko81.dat	16.29706	105828	90998	0.0004401
	sko90.dat	13.77603	131450	115534	0.0003376
	sko100a.dat	13.23272	172116	152002	0.0004113
	sko100f.dat	14.3831	170472	149036	0.0004245
	tai100a.dat	13.65246	23926646	21052466	0.0004137
	tai100b.dat	32.78682	1574846615	1.186E+09	0.0004663
	tai150b.dat	24.53774	621314634	498896643	0.0010193
	tai256c.dat	120.4809	98685678	44759294	0.0024304
	tho40.dat	30.10943	312934	240516	0.0000723
	tho150.dat	17.14005	9527466	8133398	0.0009074
	wil50.dat	11.99197	54670	48816	0.0001085
	wil100.dat	7.36674	293152	273038	0.0004091

BL:	Caso:	Desviación	Coste	Mejor Coste	Tiempo (s)
	chr22a.dat	17.28395	7220	6156	0.0029235
	chr22b.dat	12.75428	6984	6194	0.001913
	chr25a.dat	59.00949	6036	3796	0.0034377
	esc128.dat	12.5	72	64	0.3249196
	had20.dat	2.080323	7066	6922	0.002111
	lipa60b.dat	20.72294	3042381	2520135	0.0435569
	lipa80b.dat	22.18691	9486545	7763962	0.0907643
	nug28.dat	4.684475	5408	5166	0.0054675
	sko81.dat	1.652784	92502	90998	0.1684546
	sko90.dat	1.959596	117798	115534	0.2628737
	sko100a.dat	2.176287	155310	152002	0.4108217
	sko100f.dat	1.617059	151446	149036	0.3805512
	tai100a.dat	4.0102	21896712	21052466	0.1983169
	tai100b.dat	4.468294	1238989991	1.186E+09	0.4185505
	tai150b.dat	3.475323	516234924	498896643	1.528719
	tai256c.dat	0.4084336	44942108	44759294	2.95847
	tho40.dat	7.266045	257992	240516	0.0168174
	tho150.dat	2.240122	8315596	8133398	1.317498
	wil50.dat	2.765487	50166	48816	0.0301819
	wil100.dat	1.417385	276908	273038	0.3587834

Media desv greedy	54.98485
Media desv BL	9.233969
Media tiempos greedy	0.000439
Media tiempos BL	0.4262566

Como podemos ver, el algoritmo de Búsqueda Local es mucho mejor que el algoritmo Greedy.

Ésto es por que mientras el algoritmo Greedy a cada paso busca la mejor opción local desentendiéndose de las acciones ya realizadas o por realizar, y ejecutándose una y otra vez, en el algoritmo de búsqueda local los bucles internos se ejecutan una y otra vez hasta que no haya mejoras, además de que sólo compone una nueva solución si mejora a la solución anterior, por lo que es normal que obtenga mejores resultados.

En Greedy no hay correlación entre tamaño del problema y desviación típica, sin embargo en Búsqueda Local, cuanto mayor es el tamaño del problema, mejores resultados se obtienen. Ésto es por que cuantos más posibles intercambios se comprueben, más probable es que alguno mejore la solución, por lo que el algoritmo sigue funicionando.

Sin embargo, el algoritmo de búsqueda local es O(n^3), mientras que el algoritmo Greedy es O(n), por lo que el primero tarda mucho más (1000 veces más), como se ve en las tablas.

Al estar implementado en C#, no se ha usado semilla.

No se han usado parámetros específicos.

No se ha usado bibliografía externa.