Práctica 1

Problema QAP:

Técnicas de Búsqueda Local y Algoritmos Greedy 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

Práctica 1 1

- Índice 2
- Descripción del problema 3
- Consideraciones previas 3
- Algoritmos 4
- Greedy 4
- BL 4
- Procedimiento 6
- Experimento y análisis 6
- Referencias 8

Descripción del problema

Este problema consiste, a grandes rasgos, en encontrar la asignación óptima de de n unidades a n localizaciones, conociendo la distancia entre estas últimas y el flujo entre las primeras.

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} es el flujo que circula entre la unidad i y la j y d_{kl} la distancia existente entre la localización k y la l.

Para resolverlo, haremos uso de dos algoritmos probabilísticos que nos proporcionan unas soluciones bastante buenas en mucho menos tiempo.

Consideraciones previas

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.

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

```
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 = <F,MDMt>
        return coste
}
```

Para trabajar con los datos de los diferentes archivos, la función “load” cargará las matrices.

Algoritmos

Greedy

Se construye en base a una heurística que asigna las unidades más importantes (mayor intercambio de flujos) a las localizaciones más céntricas (menor distancia con el resto de localizaciones). El algoritmo en pseudocódigo sería así:

Algoritmo Greedy{

1- Calculo los vectores f y d, donde cada componente corresponde a la suma de los valores de las filas de F y D respectivamente.

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

2- Ordeno ambos vectores de menor a mayor, manteniendo guardada la información de la posiciones que ocupaban antes (posiciones reales).

3- Asigno la unidad con mayor f a la unidad con menor d:

```
pos_final = posición final del vector f
para cada componente i de d{
        resultado[f[pos_final].posicion_real] = d[i].posicion_real
        pos_final = pos_final – 1
}
```

4- Devuelvo la Solución en el vector resultado

}

Posteriormente, calculo el coste de la solución obtenida usando el algoritmo genérico.

BL

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
 }
```

Generación de soluciones aleatorias

Al comenzar el algoritmo BL, partiremos de una solución aleatoria. Para ello, inicializamos un vector v con todos los números del 0 a n-1, y posteriormente lo desordenamos usando la función random_suffle.

Si no se especifica semilla por parámetro, la función random_shuffle utilizará un generador de números aleatorios por defecto. En caso contrario, se llamará a la función myrandom, que sí hará uso de la semilla especificada.

```
si semilla no especificada
        random_shuffle(v);
en caso contrario{
        fijo la semilla
        random_shuffle( v , myrandom);
}
```

Operador de generación de vecino

Tan sencillo como cambiar el vector v por v', siempre que el resultado de la función Diferencia_Coste haya sido negativo.

```
        aux = v[i];
        v[i] = v[j];
        v[j] = aux;
```

DLB y exploración del entorno

Finalmente, el algoritmo quedaría tal cual puede verse en las trasparencias del Seminario 2, con un bucle exterior que cicla mientras el coste haya mejorado respecto a la iteración anterior.

En definitiva, el pseudocódigo al completo sería:

```
Algoritmo_BL{
        dlb[i], i ∈ (1,…,n)  = true
        v = Generar solución aleatoria
```

```
        coste = CalcularCoste(v)

        Comienzo:
                coste_anterior = coste
                diferencia = 0

                para cada posicion i de v
                        si dlb[i] == true{  // La unidad está activada
                                improve_flag = false

                                para cada posicion j de v
                                        sum = Diferencia_Coste(v,i,j)

                                        si sum < 0
                                                diferencia = sum
                                                Genero solución vecina cambiando v[i] por v[j]
                                                dlb[i]=true, dlb[j]=true // Activo ambas posiciones
                                                improve_flag=true
                        si improve_flag == false // Si no ha habido ninguna sustitución que mejore el coste
                                dlb[i]  = false // Se bloquea esta posición

                coste = coste_anterior + diferencia
        Repetir mientras: coste < coste_anterior // Repetir mientras haya mejorado
}
```

Procedimiento

La práctica ha sido desarrollada en C++, compilable por medio de un makefile (ver archivo “Leeme.txt”).

Experimento y análisis

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

	Greedy			BL		
Archivo	**Coste**	**Desviación**	**Tiempo**	**Coste**	**Desviación**	**Tiempo**
Chr22a	12138	97.17	1.3e-05	7110	15.50	0.000609
Chr22b	13280	114.40	2.5e-05	6882	11.11	0.000626
Chr25a	20414	437.78	1.0e-5	6956	83.25	0.001133
Esc128	154	140.63	8.7e-05	72	12.50	0.081065
Had20	7512	8.52	9.0e-6	6930	0.12	0.000534
Lipa60b	3218450	27.71	2.6e-05	3042515	20.73	0.009383

	Greedy			BL		
Lipa80b	10037083	29.28	3.4e-05	9474656	22.03	0.022716
Nug28	6348	22.88	1.0e-5	5360	3.76	0.00122
Sko81	105828	16.30	3.6e-05	93702	2.97	0.039952
Sko90	131450	13.78	4.9e-05	118542	2.60	0.05738
Sko100a	172116	13.23	4.9e-05	155340	2.20	0.082301
Sko100f	170472	14.38	5.0e-5	152418	2.27	0.088685
Tai100a	23936546	13.70	4.9e-05	21803948	3.57	0.055458
Tai100b	1574846615	32.79	5.1e-05	1227841036	3.53	0.085508
Tai150b	623469733	24.97	9.8e-05	519875766	4.21	0.335778
Tai256c	98685678	120.48	0.000249	45002388	0.54	0.796106
Tho40	312934	30.11	1.7e-05	248808	3.45	0.004702
Tho150	9527466	17.14	0.000104	8354932	2.72	0.341076
Wil50	54670	11.99	1.9e-05	49396	1.19	0.007646
Wil100	293152	7.37	5.0e-5	274966	0.71	0.08908

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

Algoritmo	Desviación media	Tiempo medio
Greedy	**59,73**	**~0**
BL	**9,95**	**0,11**

Como podemos ver, los resultados proporcionados por el algoritmo BL son mucho más precisos, con una desviación de menos de 10 frente a una de casi 60. En cuanto al tiempo de ejecución, el empleado por el algoritmo Greedy es ridículamente pequeño debido a su eficiencia $O(nlog(n))$. La eficiencia del algoritmo BL es sin embargo $O(n)$, pero aun así los tiempos son también bastante pequeños (aunque grandes en comparación) gracias en parte al uso de la técnica DLB y al criterio de parada.

Una vez analizados estos resultados, estamos en disposición de elegir el algoritmo que más se adecue al problema. Por lo general, si no vamos a trabajar con conjuntos de datos excesivamente grandes y/o los tiempos no son de extrema importancia, decantarse por el algoritmo BL sería razonable. Por otro lado, si el conjunto de datos es enorme y necesitamos una solución de forma rápida, deberíamos plantearnos usar el algoritmo Greedy.

También hay que destacar que, al depender de una solución aleatoria, el algoritmo BL podría variar y dar soluciones mejores y peores frente a un mismo problema, mientras que el Greedy siempre devolverá la misma.

Referencias

http://www.cplusplus.com/reference/algorithm/random_shuffle/