UNIVERSIDAD DE GRANADA

Metaheurística

Memoria de la práctica 1 de Metaheurística con el problema de asignación cuadrática (QAP)

Realizado por:
Alexander Collado Rojas

colladoalex@correo.ugr.es

Abril 2023

Índice

1. Problema de Asignación Cuadrática (QAP)

El problema de asignación cuadrática es un problema de optimización en el cual se desean asignar **N** elementos a **N** posiciones, de tal manera que se minimice la suma de los costos de la asignación hecha. Por ejemplo, se desean asignar **n** unidades a **n** localizaciones (ejemplo del hospital presentado en clases).

Para resolver el problema se provee datos como las distancias y los flujos entre las distintas unidades y localidades. Estos datos son los necesarios para poder calcular los costos de la resolución propuesta.

El problema no dispone de un algoritmo eficiente que lo pueda resolver, ya que para tamaños de entradas muy grandes los tiempos de espera también crecerían mucho. Por esa razón es un problema estudiado en metaheurística ya que se usarán algoritmos de aproximación para poder obtener soluciones cercanas a las óptimas.

2. Consideraciones del Problema

A continuación, se describirán los datos que nos proporcionan para resolver el problema y la manera en la cual se representa la solución:

2.1. Datos de Entrada

Los datos de entrada al problema serán dos matrices cuadradas de tamaño **n**. Estas matrices representan los flujos de las unidades y las distancias entre las localizaciones de las habitaciones.

$$F = \begin{pmatrix} 0 & \dots & f_{1n} \\ \vdots & \ddots & \vdots \\ f_{n1} & \dots & 0 \end{pmatrix} \qquad D = \begin{pmatrix} 0 & \dots & d_{1n} \\ \vdots & \ddots & \vdots \\ d_{n1} & \dots & 0 \end{pmatrix}$$

En cada una de las matrices se tendrá que su diagonal tendrá valor 0 ya que la distancia y el flujo entre una misma localidad/habitación es nula.

Cada una de las filas representa el flujo o la distancia entre las habitaciones. Por ejemplo, si tenemos que $f_{13} = 5$, esto significaría que el flujo entre la unidad 1 y la unidad 3 tiene un valor de 5. Y si tenemos $d_{52} = 13$, esto significaría que la distancia entre la localidad 5 y la localidad 2 tiene un valor de 13.

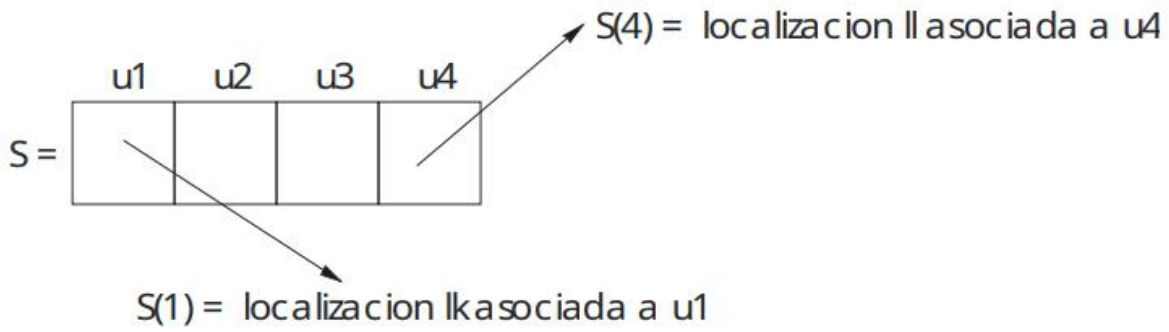
El tamaño y las matrices serán presentadas en un documento **.dat** que tiene la estructura de:

- Tamaño **n** de matrices
- Espacio en blanco
- Matriz F
- Espacio en blanco
- Matriz D

2.2. Representación de la Solución

Al inicio, para representar la solución tenía pensado una dupla en el cual la primera posición fuera la unidad y la segunda la localidad, pero a la hora de calcular el coste esto hubiera sido más complicado así que la solución será representada con un vector.

Este vector será una permutación en el cual la posición del vector indicará la unidad y el valor dentro de la posición indicará la localidad seleccionada.



2.3. Clase Reader.java

Esta clase se ha creado para poder representar los datos de entrada de un archivo **.dat** a dos matrices en **java**. Los datos serán leídos por la clase **Scanner** desde un archivo que será leído gracias a la clase **File.**

El proceso que se ha seguido es:

- A partir de la ruta proporcionada la clase File abre el archivo
- La clase Scanner lee el primer entero que determina el tamaño **n** de las matrices y lo guarda en un atributo de clase
- La clase Scanner lee los siguientes **n** enteros que corresponden a la matriz de Flujo y los va guardando en un atributo de clase que será la matriz
- La clase Scanner lee los últimos **n** enteros que corresponden a la matriz de Distancia y la guardará en otra matriz.

Los demás métodos son solo consultores para que las otras clases puedan copiar/leer los valores de las matrices y tamaños.

2.4. Método de Coste

La función que calculará el coste será:

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

Se tendrá que multiplicar cada uno de los elementos del flujo por la respectiva distancia de la solución propuesta.

```
Inicio
      Funcion calcularCosteSolucion(matrizFlujo, matrizDistancia, vectorPermutado)
            costo <- 0

            Para cada fila de vectorPermutado
                  l <- vectorPermutado[i]

                  Para cada columna de vectorPermutado
                        Si j es diferente de i entonces
                              k <- vectorPermutado[j]
                              costo <- costo + matrizFlujo[i][j] * matrizDistancia[l][k]
                        Fin Si
                  Fin Para
            Fin Para

            Devolver costo

      Fin Funcion
Fin
```

2.5. Método de Generar Permutación Aleatoria

Método necesario para poder ejecutar el algoritmo de **búsqueda local**. Se necesita generar un vector permutado aleatoriamente de tamaño **n**. Este vector simulará una solución aleatoria, ningún valor se puede repetir.

Dentro de la función hay una función **contiene** esta solo verifica que el valor entero aleatorio generado no esté en vectorGenerado, es decir que no se repita

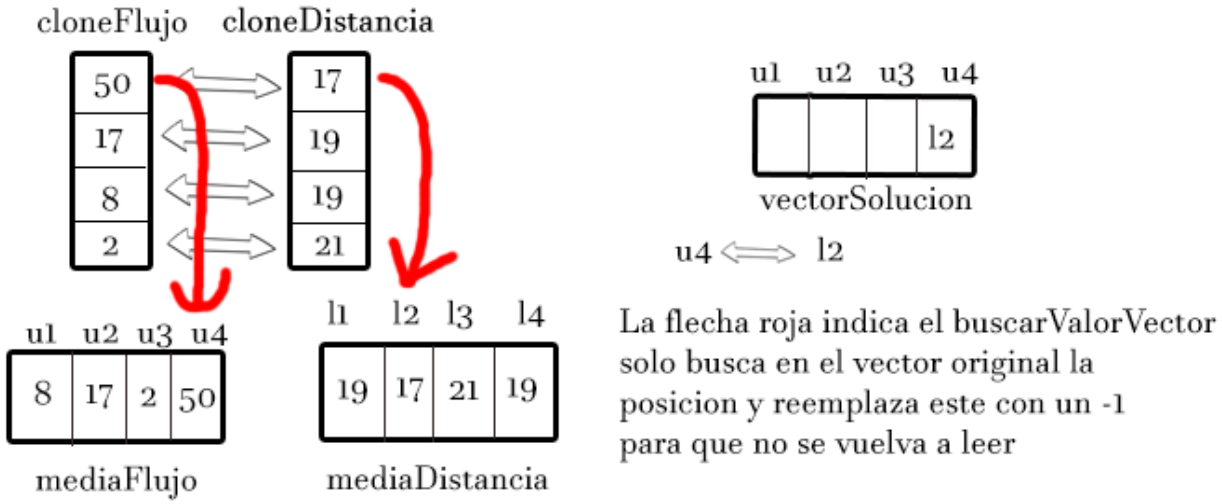
```
Inicio
	Funcion generarVectorAleatorio(n)
		vectorGenerado <- vector de tamaño n
			random <- nueva instancia de la clase Random

		Desde i hasta n
			valor <- 0

			Mientras contiene(vectorGenerado, valor) hacer
				valor <- nuevo entero random menor que n
			Fin Mientras

			vectorGenerado[i] <- valor
		Fin Para

		Para i hasta n
			//El menos uno es para que sean valores de 0 hasta n
			vectorGenerado[i] <- vectorGenerado[i] - 1
		Fin Para

		Devolver vectorGenerado

	Fin Funcion
Fin
```

3. Algoritmo Greedy

El algoritmo Greedy será representado en una clase con atributos de instancia como pueden ser: **matrices de entrada para el flujo y las distancias, vectores para las medias de estos dos valores y un vector solución.**

La idea del algoritmo es la siguiente:

- Hacer una media por filas de ambas matrices y almacenar estas en sus vectores correspondientes.
- Clonar estos vectores para así poder ordenarlos de mayor a menor y de menor a mayor.
- Ahora con los vectores ordenados solo queda parear los valores de mayor flujo con menor distancia y para esto se buscará en el vector de medias original cual es la posición para así asignársela en el vector solución. Por ejemplo:



El método **arreglarVectorSolucion** solo pasa de la dupla que había considerado al inicio a un vector solución permutado.

Por último, el método **calcularMediaFila(matriz)** solo hace la media de cada una de la fila de una matriz proporcionada por parámetros.

El pseudocódigo correspondiente al **Algoritmo Greedy** es el siguiente:

```
Inicio

      n <- tamaño de matrices

      Funcion solucionGreedy()

            //Generar vectores de medias
            mediaDistancia <- calcularMediaFila(matrizDistancia)
            mediaFlujo <- calcularMediaFila(matrizFlujo)


            solucion <- nuevo vector de dupla de tamaño n

            //Clonar los vectores de media
            cloneDistancia <- clonar(mediaDistancia)
            cloneFlujo <- clonar(mediaFlujo)

            //Mayor a menor
            Ordenar(cloneFlujo)

            //Menor a mayor
            Ordenar(cloneDistancia)

            //Parear los valores de mayor flujo con menor distancias
            Para i hasta n hacer
                  flujo <- cloneFlujo[i]
                  posicionFlujo <- buscarValorVector(flujo, mediaFlujo)

                  distancia <- cloneDistancia[i]
                  posicionDistancia <- buscarValorVector(distancia, mediaDistancia)

                  solucion[i] <- nuevo par(posicionFlujo, posicionDistancia)
            Fin Para

      arreglarVectorSolucion()

      Devolver solucionDefinitiva
Fin Funcion
```

Así de esta manera se busca obtener una primera aproximación a una solución que no es óptima, pero es sencilla y eficiente con los tiempos de ejecución. Las unidades que mayor flujo tienen serán las que necesiten localidades más cercanas es decir con menores distancias.

4. Algoritmo de Búsqueda Local

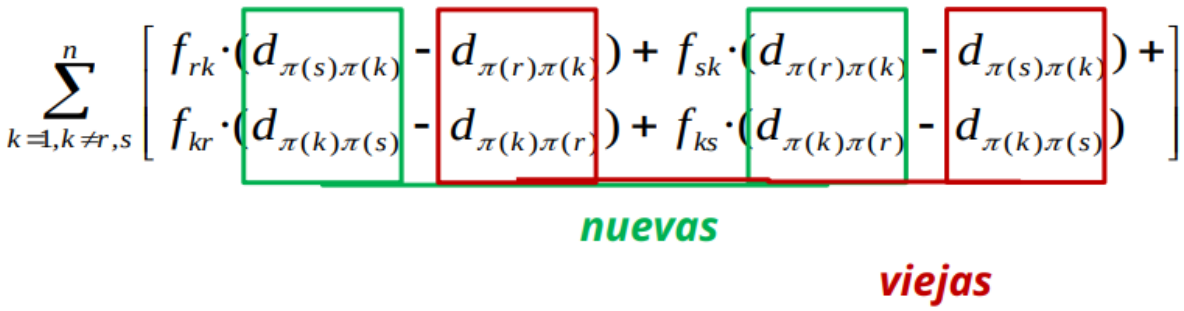
El algoritmo de búsqueda local será representado en una clase en la cual tendremos como atributos de instancias:

- **Matrices de flujo y distancia**
- **Vector permutado**: A partir de este vector generado aleatoriamente o mediante una semilla se buscarán nuevas soluciones y se comprobará si estas mejoran o empeoran el coste.
- **Vector binario:** Este vector binario será en el cual emplearemos la técnica de **DLB** así evitando que si un intercambio empeora entonces el sistema no lo vuelva a tener en cuenta. Al inicio tendrá todas sus posiciones a falso.

A continuación, se describirán los demás métodos usados

4.1. Factorización del Coste

Para la creación de este método solo quedaría pasar esta función del seminario a código, en la cual da como resultado la diferencia entre la posición antes de intercambiar y luego de intercambiar. Si la solución es negativa entonces esto significará que este intercambio es mejor y se partirá ahora de esta nueva permutación para seguir buscando nuevas y mejores soluciones




```
Inicio
función factorizaBL(pos1, pos2):
    coste = 0
    para i hasta n hacer:
        si i no es igual a pos1 y i no es igual a pos2 entonces:
            coste += (matrizFlujo[pos1][i] * (matrizDistancia[vectorPermutado[pos2]][vectorPermutado[i]] - matrizDistancia[vectorPermutado[pos1]][vectorPermutado[i]]) +
                    matrizFlujo[pos2][i] * (matrizDistancia[vectorPermutado[pos1]][vectorPermutado[i]] - matrizDistancia[vectorPermutado[pos2]][vectorPermutado[i]]) +
                    matrizFlujo[i][pos1] * (matrizDistancia[vectorPermutado[i]][vectorPermutado[pos2]] - matrizDistancia[vectorPermutado[i]][vectorPermutado[pos1]]) +
                    matrizFlujo[i][pos2] * (matrizDistancia[vectorPermutado[i]][vectorPermutado[pos1]] - matrizDistancia[vectorPermutado[i]][vectorPermutado[pos2]]))
    devolver coste
Fin
```

4.2. Método aplicarMovimiento

Este método será el que aplique los movimientos de intercambio de las posiciones para buscar una nueva y mejor permutación. Este método además le suma al coste de la permutación la factorización del cambio de las dos posiciones.

4.3. Búsqueda Local

Para la realización del algoritmo de la búsqueda local usando el **DLB** solo ha sido necesario implementar uno de los pseudocódigos del seminario (diap. 23) el incluyéndole los bucles de las 50000 iteraciones o hasta que no se encuentren unas soluciones mejores.

```
procedure iterative improvement
    for i = 1 to n do
      if dlb[i] = 0 then
        improve_flag ← false
        for j = 1 to n do
            CheckMove(i, j)
            if move improves then
                ApplyMove(i, j); dlb[i] ← 0, dlb[j] ← 0
                improve_flag ← true
        endfor
        if improve_flag = false then dlb[i] ← 1
    end
end iterative improvement
```

IMPORTANTE: Este es el bucle interno de la BL, el de exploración del vecindario de la solución actual. Sea cual sea la BL usada, siempre habrá un bucle externo que repetirá el proceso mientras se produzca mejora

23

```
Funcion algoritmoBL()
      improve_flag <- verdadero
      mejoraEntorno <- verdadero
      mejoraCoste <- falso

      iteracionesMaximas <- 50 000

      Para k desde 0 hasta iteracionesMaximas y mejoraEntorno hacer
            mejoraEntorno <- falso

            //Bucle interno exploracion del vecindario -- Diapositivas Seminario
            Para i desde 0 hasta tamanioVector hacer
                  Si vectorBinario[i] es falso entonces
                        improve_flag <- falso

                        Para j desde 0 hasta tamanioVector hacer
                              //Se verifca si el valor de factorización es negativo es decir que mejora
                              mejoraCoste <- (factorizaBL(i, j) < 0)
                              Si mejoraCoste entonces
                                    aplicarMovimiento(i, j, factorizaBL(i, j))
                                    vectorBinario[i] <- falso
                                    vectorBinario[j] <- falso
                                    improve_flag <- verdadero
                                    mejoraEntorno <- verdadero //Se encontró una mejora en el entorno
                              Fin Si
                        Fin Para
                        Si no improve_flag entonces
                              vectorBinario[i] <- verdadero
                        Fin Si
                  Fin Si
            Fin Para
      Fin Para

      Devolver vectorPermutado
Fin Funcion
```

El algoritmo de búsqueda local comienza con una solución aleatoria que será una permutación generada aleatoriamente por el método ya descrito en el apartado 2 y se realizarán iteraciones en las que se exploran las soluciones vecinas para encontrar una mejor solución. La solución actual se mejora a veces si por ejemplo se intercambian dos elementos, hasta que se alcanza un mínimo local, es decir, una solución que no puede mejorarse mediante movimientos locales.

5. Concepto del desarrollo de la Práctica

5.1. Lenguaje del Proyecto

La práctica 1 ha sido desarrollada únicamente utilizando **Java** utilizando Netbeans como entorno de desarrollo. Se ha optado por crear un proyecto **QAP** que tendrá ciertas clases como pueden ser:

- **Reader.java:** De esta clase se leerán los valores de entrada, es decir, las matrices y su tamaño. También es utilizada para nombrar a todos los archivos y que el usuario pueda elegir cual desee ejecutar.
- **QAP.java:** En esta clase está el **main** del proyecto y además métodos de uso común como el coste.
- **Greedy.java:** Se desarrolla el algoritmo Greedy.
- **BL.java:** Se desarrolla el algoritmo de Busqueda Local, factorización y aplicar movimiento
- **Pair.java:** Para el inicio del proyecto, simulando que el vector solución era una dupla.

5.2. Estructuración del Proyecto

El proyecto a entregar seguirá la siguiente estructura. Como aclaración para no copiar todos los archivos de entrada los ejemplifico como ***.dat**.

```
.Proyecto/
      |-- MemoriaMetaheuristica.pdf
      |-- software/
            |-- FUENTES/
            |     |-- qap/
            |           |-- *.java
            |-- BIN/
            |     |-- qap/
            |           |-- *.class
            |-- Tablas/
                  |-- *.dat
```

5.3. Ejecución del Proyecto

Si bien he utilizado **Netbeans** para la ejecución del proyecto también se puede hacer de forma manual de la siguiente manera:

- Asegurarse de tener el kit de desarrollo (JDK) instalado
- Compilar todos los archivos **.java** utilizando el comando **javac**

```
alumno@colladoalex:~/Escritorio/MH/Software$ javac -d BIN FUENTES/qap/*.java
Note: FUENTES/qap/Greedy.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
alumno@colladoalex:~/Escritorio/MH/Software$
```

- Ya solo queda situarte en la carpeta BIN/ y ejecutar **java qap/QAP.** Que sería ejecutar la clase en la cual se encuentra el main. Es necesario estar en BIN/ porque sino no reconoce QAP como main class.

```
alumno@colladoalex:~/Escritorio/MH/Software$ cd BIN
alumno@colladoalex:~/Escritorio/MH/Software/BIN$ java qap/QAP
0 - Algoritmo Greedy
1 - Algoritmo Busqueda Local

Elige cual algoritmo quieres usar:
0
Elige el archivo que desees usar:
chr22a.dat

El coste de la solucion es: 13538
Tiempo de ejecucion: 6263 microsegundos
Permutacion Solucion:
ARCHIVO ELEGIDO: chr22a.dat
10, 21, 17, 19, 13, 11, 2, 14, 22, 6, 9, 18, 20, 5, 3, 1, 12, 16, 7, 15, 8, 4,
```

5.3.1. Problemas de Ejecución

Mientras ejecutaba por el terminal he tenido que cambiar el formato de entrada de archivo, en netbeans por ejemplo listaba todos los archivos para que el usuario solo tomara el número del que quería ejecutar, pero por el siguiente error se ha cambiado para que el usuario escriba el nombre del archivo con todo y extensión.

```
Elige el archivo que desees usar:
Exception in thread "main" java.lang.NullPointerException: Cannot read the array
 length because "<local4>" is null
        at qap.Reader.obtenerTodosArchivos(Reader.java:102)
        at qap.QAP.main(QAP.java:37)
```

6. Experimento y análisis de los resultados

Para evaluar los algoritmos implementados tanto el **Greedy** como el de **Búsqueda Local** se utilizarán los valores de tiempo y coste. Los valores de costes se compararán con los proporcionados en **Tablas_QAP_2022-23** para sacar su valor de **Fitness**. Luego de tener todos los valores de Fitness y Tiempo se hará una media de estos para cada uno de los algoritmos.

Los tiempos han sido tomados en nanosegundos, luego estos serán divididos entre 1000 para convertirlos en microsegundos que será lo que salga por pantalla. Para las tablas estos los he convertido en milisegundos.

Aclaración: Los tiempos han sido medidos ejecutando el programa en Linux mediante una máquina virtual así que esto también podría influir.

Algoritmo Greedy			
Caso	**Coste obtenido**	**Fitness**	**Tiempo**
Chr22a	13538	119.92	13.25
Chr22b	11942	92.80	12.98
Chr25a	17556	362.49	14.14
Esc128	142	121.88	14.08
Had20	7622	10.11	7.53
Lipa60b	3232061	28.25	9.42
Lipa80b	10047868	29.42	8.50
Nug28	6152	19.09	6.25
Sko81	105828	16.30	12.55
Sko90	131450	13.78	15.51
Sko100a	172116	13.23	13.23
Sko100f	170472	14.38	11.35
Tai100a	23926646	13.65	13.66
Tai100b	1574846615	32.79	9.05
Tai150b	621314634	24.54	15.22
Tai256c	98685678	120.48	25.45
Tho40	312934	30.11	7.69
Tho150	9527466	17.14	26.35
Wil50	54670	11.99	12.00
Wil100	293152	7.37	13.44

Algoritmo Búsqueda Local - Con Semilla			
Caso	**Coste obtenido**	**Fitness**	**Tiempo**
Chr22a	6992	13.58	8.31
Chr22b	7678	23.96	5.61
Chr25a	5838	53.79	22.35
Esc128	68	6.25	74.34
Had20	6982	0.87	5.90
Lipa60b	2520135	0.00	46.98
Lipa80b	9471466	21.99	85.75
Nug28	5390	4.34	9.33
Sko81	93338	2.57	72.84
Sko90	118548	2.61	89.76
Sko100a	154994	1.97	80.85
Sko100f	152042	2.02	84.70
Tai100a	21920984	4.13	70.87
Tai100b	1222743677	3.10	91.04
Tai150b	518741546	3.98	151.72
Tai256c	45029300	0.60	417.32
Tho40	248424	3.29	32.06
Tho150	8378974	3.02	145.37
Wil50	49542	1.49	56.47
Wil100	276562	1.29	81.80

Para medir los valores de costo de Búsqueda Local y que no sean siempre aleatorios se ha puesto como semilla la permutación solución del Greedy.

Valores Medios:

Greedy	
Media Fitness:	**54.98**
Media Tiempo:	**13.08**

Busqueda Local	
Media Fitness:	**7.74**
Media Tiempo:	**81.67**