
UNIVERSIDAD DE GRANADA


# *Metaheurística*

*Memoria de la práctica 3 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.

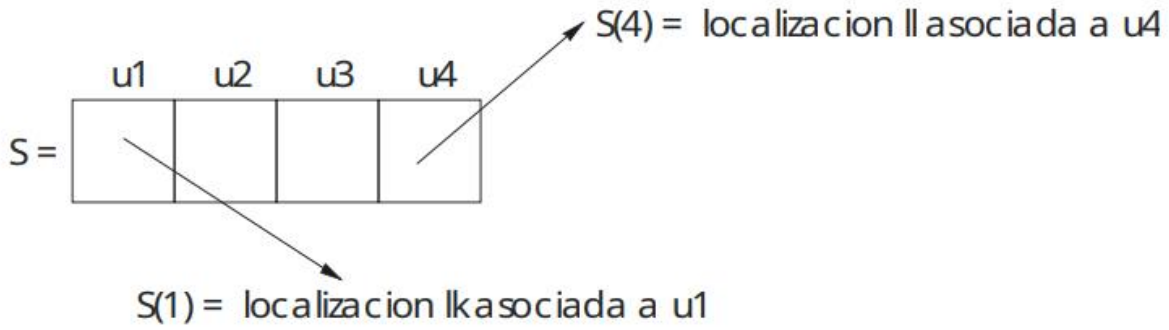
El tamaño y las matrices serán presentadas en un documento **.dat** que tiene la estructura de:

- 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. Búsqueda Basadas en Trayectorias

Las técnicas de búsqueda basadas en trayectorias son algoritmos utilizados para encontrar una solución óptima o aproximada en problemas de búsqueda en espacios de estados. Estas técnicas se basan en la idea de explorar y seguir una "trayectoria" o secuencia de acciones desde un estado inicial hasta un estado objetivo.

## 3.1. Enfriamiento Simulado

El Enfriamiento o Recocido Simulado es un algoritmo de búsqueda por entornos con un criterio probabilístico de aceptación de soluciones basado en Termodinámica.

Un modo de evitar que la búsqueda local finalice en óptimos locales, hecho que suele ocurrir con los algoritmos tradicionales de búsqueda local, es permitir que algunos movimientos sean hacia soluciones peores. Pero si la búsqueda está avanzando realmente hacia una buena solución, estos movimientos "de escape de óptimos locales" deben realizarse de un modo controlado

Esto se realiza controlando la frecuencia de los movimientos de escape mediante una función de probabilidad que hará disminuir la probabilidad de estos movimientos hacia soluciones peores conforme avanza la búsqueda (y por tanto estamos más cerca, previsiblemente, del óptimo local)

### Atributos Importantes

#### Atributos de Instancia

- Las respectivas matrices de **flujo** y **distancia** junto con su respectivo tamaño que tendrá el vector solución.
- Atributo de tipo **Random** esto para poder realizar ejecuciones con una semilla. Para este y los demás problemas la semilla **random** será **42**.
- Constantes definidas en la descripción de la práctica.
    - **MU (μ)** estará fijada a 0.3.
    - **PHI (ϕ)** estará fijado a 0.2 que es la probabilidad de aceptar una solución.
    - **MAX_EV** será el máximo de evaluaciones que el algoritmo podrá realizar, este máximo se fija a 50 000.

- Instancia de la clase **QAP** para así poder llamar a métodos comunes entre los algoritmos como puede ser **generarVectorAleatorio** o **calcularCosteSolucion.**
- Vector **mejorSolución** que será aquel el cual devolverá el algoritmo.
- Vector **s** como vector intermedio para realizar los intercambios y demás
- Variables **max_vecinos** y **max_exitos** definidas en el guión de prácticas como condición de salida de uno de los bucles del algoritmo.
- **Beta:** Fórmula respectiva con el esquema de enfriamiento utilizando el esquema de Cauchy.

$$\beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

**Variables Importantes**

- **T:** Temperatura en un momento dado del problema, seguirá la siguiente fórmula.

$$T_{k+1} = \frac{T_k}{1 + \beta \cdot T_k}$$

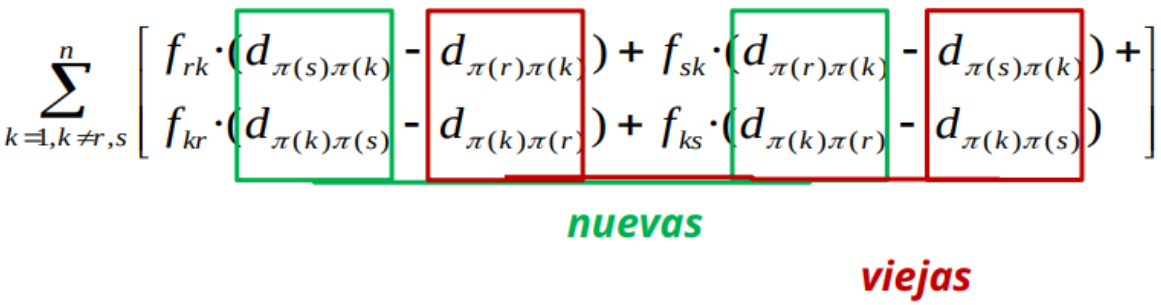
- **T0:** Temperatura inicial del problema, se calcula con la siguiente fórmula.

$$T_0 = \frac{\mu \cdot C(S_0)}{-\ln(\varphi)}$$

- **Tf:** Temperatura final, esta siempre deberá ser menor que la temperatura inicial. Se fija al inicio del algoritmo a 0.0001, es decir, 10 elevado a -4.
- **C(S0):** Es el coste de la solución inicial

## Factorización

Se utilizará el mismo método de factorización que el de la **Búsqueda Local** de la práctica 1. Donde si el resultado de este método es negativo significa que el cambio realizado hace que mejore la solución.




## Operador de Vecino (Intercambio)

Este operador será el que realice el intercambio de entre dos posiciones aleatorias del vector. Se escogerán aleatoriamente las unidades r y s a las cuales se les cambiará la localización.

```
Inicio
    función intercambioES(vector, i, j)
        auxiliar
        vectorReturn <- copiar(vector)

        auxiliar <- vectorReturn[i]
        vectorReturn[i] <- vectorReturn[j]
        vectorReturn[j] <- auxiliar

        devuelve vectorReturn
    Fin función
Fin
```

## Algoritmo del Enfriamiento Simulado

Se realizó siguiendo el pseudocódigo presentado en las diapositivas.

```
Procedimiento Simulated Annealing (Δf para minimizar)
Start
 T ←T0;   s ← GENERATE();   Best Solution ← s;
 Repeat
     For cont = 1 to L(T) do /* Inner loop
      Start
     S' ←  NEIGHBORHOOD_OP(s); /* A single move
      Δf=f(s')–f(s);
      If ((Δf<0) or (U(0,1) <= exp(-Δf/k·T))) then
        S ← s';
                If COST(s) is better than COST(Best Solution)
         then Best Solution ←  s;
       End
    T ← g(T); /* Cooling scheme. The classical one is geometric: T ← α·T
 until (T ←  Tf); /* Outer loop
 Return(Best Solution);
End
```

La primera parte del algoritmo consiste en generar una solución inicial aleatoria y asignarle esta a la **mejorSolucion** a partir de esto calcular el **T0** y asignarlo a **T**. Además de declarar algunas variables para poder controlar los accesos a los bucles y los criterios de parada como pueden ser **exitosEnfriamientos o vecinos.** Se comprueba siempre que T0 sea mayor que Tf

En este punto se entra en el bucle **do-while** que consiste en repetirlo hasta que la temperatura actual sea menor que la temperatura fnal. Se inicializan **vecinos y éxitosEnfriamientos** a 0 para poder comprobarlos en el siguiente bucle.

En el bucle **while** mientras no se cumpla la condición se realizará un **intercambio** entre las posiciones del vector generado al inicio del problema al cual se le deberá calcular la **factorización,** se aumentan el contador de vecino ya que se está explorando otras áreas. Luego en un condicional se verifica si el coste ha mejorado, es decir, que sea un coste negativo o si la exponencial de **la factorizacaion entre la temperatura actual** sea mayor o igual a la probabilidad **U** comprendida entre 0 y 1. Si alguna de estas dos condiciones se cumple entonces hay un éxito en el enfriamiento y se pasa a comprobar los costes del vector **s** con el vector **mejorSolucion** donde si mejora entonces este pasa a ser la nueva **mejorSolucion**.

## Criterio de parada

Condición de parada: El algoritmo finalizará bien cuando haya alcanzado el número máximo de evaluaciones prefijado o bien cuando el número de éxitos en el enfriamiento actual sea igual a 0.

Otra condición del bucle **do-while** es si **T** es igual o menor a **Tf.**

## Pseudocódigo

```
Función ES() -> arreglo de enteros
    sp <- nuevo arreglo de enteros con tamaño tamanioMatriz
    T0 <- 0.0
    Tf <- 0.0001  // 10e-4
    T <- 0.0

    costeS <- 0
    costeBest <- 0
    t <- 0

    mejoraCoste <- 0

    mejorSolucion <- s

    costeMejorSolucion <- calcularCosteSolucion(this.matrizFlujo, this.matrizDistancia, this.mejorSolucion)

    T0 <- (MU * costeMejorSolucion) / (-log(PHI)) //Temperatura Inicial

    // TF siempre debe ser menor que T0, teniendo un valor muy cercano a 0
    Si T0 < Tf entonces
        Tf <- T0 * Tf
    Fin Si

    T <- T0

    Beta <- (T0 - Tf) / (T0 * Tf * numeroIteraciones)

    Hacer
        exitosEnfriamiento <- 0
        vecinos <- 0
```

```
    Mientras exitosEnfriamiento < max_exitos y vecinos < max_vecinos hacer
        Haccer
            i <- generarEnteroAleatorio(0, tamanioMatriz)
            j <- generarEnteroAleatorio(0, tamanioMatriz)
        Mientras i == j

        // Factorizacion
        mejoraCoste <- factorizar(s, i, j)
        t++

        //Vecino / Intercambio
        sp <- intercambioES(s, i, j)
        vecinos++

        U <- obtenerDecimalAleatorio() //Entre 0 y 1

        // K = 1. Asi que T * K = T
        Si (mejoraCoste < 0) o (U <= exp(-mejoraCoste / T)) entonces
            s <- sp
            exitosEnfriamiento++

            costeS <- calcularCosteSolucion(this.matrizFlujo, this.matrizDistancia, s)
            costeBest <- calcularCosteSolucion(this.matrizFlujo, this.matrizDistancia, this.mejorSolucion)
            t++

            Si costeS < costeBest entonces
                mejorSolucion <- s
            Fin Si
        Fin Si
```

```
        Fin mientras

        Si exitosEnfriamiento == 0 entonces
            romper //Salirse del bucle
        Fin Si

        T <- T / (1 + Beta * T)

    Mientras T > Tf y t < MAX_EV

    devoler mejorSolucion
Fin función
```

## 3.2. Búsqueda Local Modificada

Algoritmo de búsqueda local: En ILS y VNS, se considerará la búsqueda local (BL) que sigue el enfoque del primer mejor vecino, pero escogiendo las posiciones de forma aleatoria y sin aplicar la máscara de bits. En cada iteración del bucle se aplicará un único movimiento Int(□,r,s) para generar una única solución vecina que será comparada con la solución actual. Se escogerán aleatoriamente las unidades r y s a las cuales se les cambiará la localización. No se considerará el uso de la máscara DLB de la BL de la Práctica 1.a. Se detendrá la ejecución del algoritmo cuando se hayan evaluado el número máximo de soluciones.

Se tienen dos métodos para este algoritmo, el primero sin parámetros para utilzarlo en **ILS** y el segundo con el parámetro **k** para el **VNS** indicando cuantos aleatorios se generan.

## Pseudocódigo

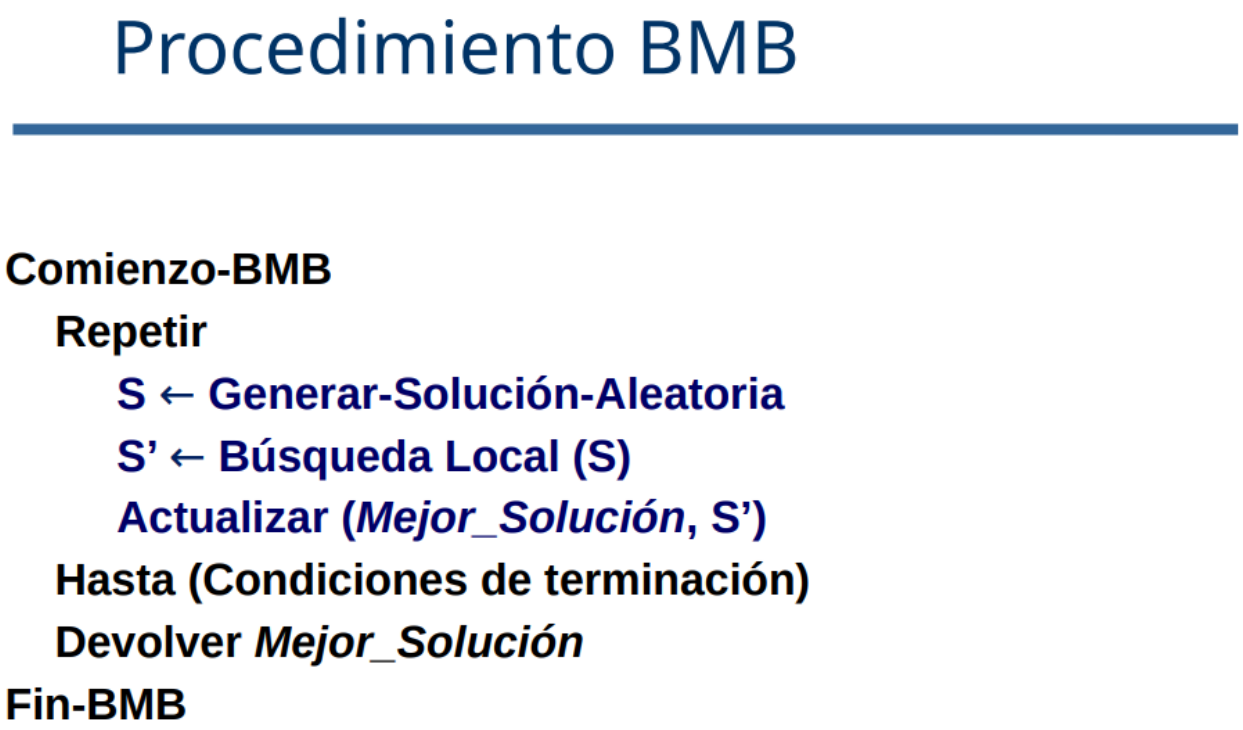
```
Función algoritmoBL() -> arreglo de enteros
    mejoraCoste <- falso


    Para k desde 0 hasta iteracionesMaximas hacer

        Hacer
            i = obtenerEnteroAleatorioEnRango(0, tamanioVector)
            j = obtenerEnteroAleatorioEnRango(0, tamanioVector)
        Mientras i == j

        mejoraCoste = (factorizaBL(i, j) < 0)
        Si mejoraCoste entonces
            aplicarMovimiento(i, j, factorizaBL(i, j))
        Fin Si

    Fin para

    devolver vectorPermutado
Fin función
```

```
Función algoritmoBL(k_vns: entero) -> arreglo de enteros
    mejoraCoste <- falso


    Para k desde 0 hasta iteracionesMaximas hacer

        Para t desde 0 hasta k_vns hacer
            Hacer
                i = obtenerEnteroAleatorioEnRango(0, tamanioVector)
                j = obtenerEnteroAleatorioEnRango(0, tamanioVector)
            Mientras i == j

            mejoraCoste = (factorizaBL(i, j) < 0)
            Si mejoraCoste entonces
                aplicarMovimiento(i, j, factorizaBL(i, j))
            Fin Si

    Fin para

    devolver vectorPermutado
Fin función
```

## 3.3. Búsqueda Multiarranque Básica

El algoritmo BMB consistirá simplemente en generar un determinado número de soluciones aleatorias iniciales y optimizar cada una de ellas con el algoritmo de BL indicado. Se devolverá la mejor solución encontrada en todo el proceso.

Se implementó siguiendo el pseudocódigo propuesto en las diapositivas del seminario:




**Atributos de Instancia**

- Las respectivas matrices de **flujo** y **distancia** junto con su respectivo tamaño que tendrá el vector solución.
- Atributo de tipo **Random** esto para poder realizar ejecuciones con una semilla. Para este y los demás problemas la semilla **random** será **42**.
- **Instancia** de la clase de Búsqueda Local (de la práctica 1) para poder hacer la llamada a **algoritmoBL().**
- **Instancia** de la clase **QAP** para poder hacer la llamada a **generarVectorAleatorio()** y **calcularCosteSolucion().**
- **mejorSolucion** vector que irá almacenando posibles soluciones.

- **Random**, atributo utilizado para fijar semillas y que las soluciones sean siempre las mismas siempre y cuando se elija la opción de semilla.

**Pseudocódigo**

```
Función BMB():
    iteraciones <- 25
    costeA <- 0
    costeB <- valor máximo de entero
    vectorSolucionAux <- arreglo de enteros de tamaño tamanioMatriz

    Para i = 0 hasta iteraciones - 1:

        // Generar solución aleatoria
        mejorSolucion <- generarVectorAleatorio(tamanioMatriz, random)

        instanciaBL <- BL(mejorSolucion, matrizFlujo, matrizDistancia, 2000)

        // Aplicar BL (De la P1) a la solución generada anteriormente
        mejorSolucion <- instanciaBL.algoritmoBL()

        // Actualizar solución respecto al coste
        costeA <- calcularCosteSolucion(matrizFlujo, matrizDistancia, mejorSolucion)

        Si costeA < costeB entonces:
            vectorSolucionAux <- mejorSolucion
            costeB <- costeA
        Fin Si
    Fin para


    devolver vectorSolucionAux
Fin funcion
```

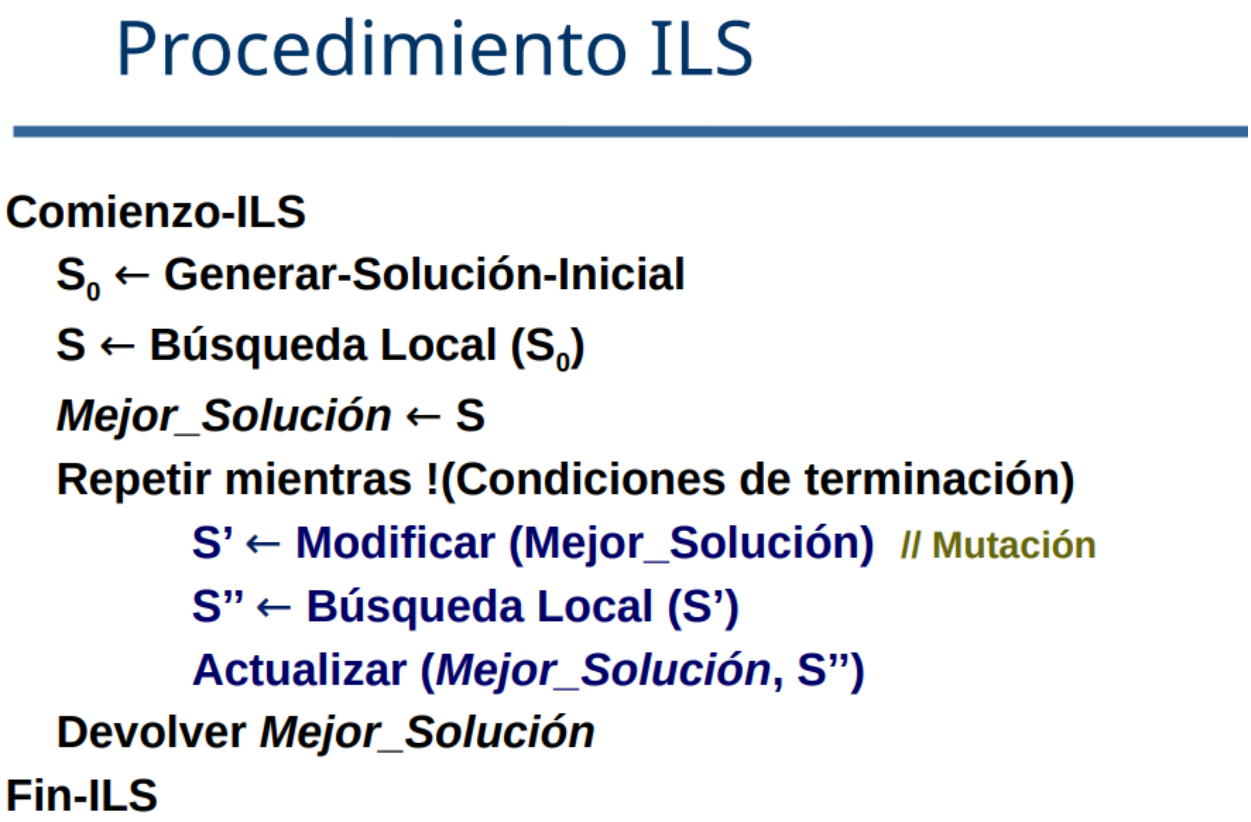
## 3.4. Búsqueda Local Reiterada

El algoritmo ILS consistirá en generar una solución inicial aleatoria y aplicar el

algoritmo de BL sobre ella. Una vez obtenida la solución optimizada, se estudiará si es mejor que la mejor solución encontrada hasta el momento y se realizará una mutación sobre la mejor de estas dos, volviendo a aplicar el algoritmo de BL sobre esta solución mutada. Este proceso se repite un determinado número de veces, devolviéndose la mejor solución encontrada en todo el proceso. Por tanto, se sigue el criterio del mejor como criterio de aceptación de la ILS.

Tal y como se describe en las transparencias del Seminario 4, el operador de mutación de ILS estará basado en un operador de vecino que provoque un cambio más brusco en la solución actual que el considerado en la BL. Para ello, usaremos el operador de modificación por sublista aleatoria de tamaño fijo t. Este proceso consiste en generar aleatoriamente una posición i de inicio de la sublista y reasignar aleatoriamente el orden de los nodos existentes entre esa posición i y la posición i+t (de forma circular).

Se implementó siguiendo el pseudocódigo de las transparencias del seminario:




**Atributos de Instancia**

- Las respectivas matrices de **flujo** y **distancia** junto con su respectivo tamaño que tendrá el vector solución.
- Atributo de tipo **Random** esto para poder realizar ejecuciones con una semilla. Para este y los demás problemas la semilla **random** será **42**.
- **Instancia** de la clase de Búsqueda Local (la modificada) para poder hacer la llamada a **algoritmoBL().**

- **Instancia** de la clase **QAP** para poder hacer la llamada a **generarVectorAleatorio()** y **calcularCosteSolucion().**
- **mejorSolucion** vector que irá almacenando posibles soluciones.
- **Random**, atributo utilizado para fijar semillas y que las soluciones sean siempre las mismas siempre y cuando se elija la opción de semilla.

### Método mutarVector()

- Se crea una copia del vector original utilizando el método "clone()" para evitar modificar el vector original.
- Se genera un número aleatorio para seleccionar una posición aleatoria en el vector.
- Se calcula el valor de "t" dividiendo la longitud del vector por 3. Esto determina cuántos elementos se modificarán a partir de la posición aleatoria.
- Se crea una lista llamada "elementosModificar" para almacenar los elementos que se modificarán. Estos elementos son desde la posición aleatoria hasta posición aleatoria + t.
- La lista "elementosModificar" se mezcla aleatoriamente
- Se insertan los “elementosModificar” al vector original
- Finalmente, se devuelve la copia del vector modificado.

```
Función mutarVector(vector): arreglo de enteros
    auxVec <- clonar(vector)
    posicion <- generarEnteroAleatorio(0, longitud(auxVec) - 1)  // Seleccionar una posición aleatoria en el vector
    t <- longitud(auxVec) / 3  // Determinar cuántos elementos modificar a partir de la posición

    elementosModificar = nueva Lista de enteros

    Para i = 0 hasta t - 1:
        elementosModificar.agregar(auxVec[(posicion + i) % longitud(auxVec)])  // Modo cíclico
    Fin para

    mezclarLista(elementosModificar)

    Para i = 0 hasta t - 1:
        auxVec[(posicion + i) % longitud(auxVec)] = elementosModificar[i]
    Fin para

    devolver auxVec
Fin funcion
```

### Pseudocódigo

```
Función ILS(): arreglo de enteros
    costeA <- 0
    costeB <- valor máximo de entero

    vectorIntermedio <- nuevo arreglo de enteros de tamaño tamanioMatriz

    // Generar vector aleatorio inicial
    vectorIntermedio <- generarVectorAleatorio(tamanioMatriz, random)

    BLm <- BL_Modificado(matrizFlujo, matrizDistancia, tamanioMatriz, random, vectorIntermedio, 2000)

    // Aplicar Búsqueda Local (BL) al vector aleatorio inicial
    mejorSolucion <- BLm.algoritmoBL()

    Para i = 0 hasta 24 - 1:
        // Mutación
        vectorIntermedio <- mutarVector(this.mejorSolucion)
        BLm <- BL_Modificado(matrizFlujo, matrizDistancia, tamanioMatriz, random, vectorIntermedio, 2000)
        vectorIntermedio <- BLm.algoritmoBL()

        costeA <- calcularCosteSolucion(matrizFlujo, matrizDistancia, vectorIntermedio)

        Si costeA < costeB entonces:
            this.mejorSolucion <- vectorIntermedio
            costeB <- costeA
        Fin si
    Fin para

    devolver this.mejorSolucion
Fin funcion
```

## 3.5. Hibridación entre Búsqueda Local Reiterada y Enfriamiento Simulado

Será idéntico al anterior exceptuando que cada llamada que se hace en vez de hacerse a la búsqueda local **modificad** se realizará al algoritmo **ES**.

El algoritmo ILS consistirá en generar una solución inicial aleatoria y aplicar el algoritmo de ES sobre ella. Una vez obtenida la solución optimizada, se estudiará si es mejor que la mejor solución encontrada hasta el momento y se realizará una mutación sobre la mejor de estas dos, volviendo a aplicar el algoritmo de ES sobre esta solución mutada. Este proceso se repite un determinado número de veces, devolviéndose la mejor solución encontrada en todo el proceso. Por tanto, se sigue el criterio del mejor como criterio de aceptación de la ILS.

### Atributos de Instancia

- Las respectivas matrices de **flujo** y **distancia** junto con su respectivo tamaño que tendrá el vector solución.

- Atributo de tipo **Random** esto para poder realizar ejecuciones con una semilla. Para este y los demás problemas la semilla **random** será **42**.
- **Instancia** de la clase **ES** para hacer la llamada a este algoritmo.
- **Instancia** de la clase **QAP** para poder hacer la llamada a **generarVectorAleatorio()** y **calcularCosteSolucion().**
- **mejorSolucion** vector que irá almacenando posibles soluciones.
- **Random**, atributo utilizado para fijar semillas y que las soluciones sean siempre las mismas siempre y cuando se elija la opción de semilla.

### Pseudocódigo

```
Función ILS_ES(): arreglo de enteros
    costeA <- 0
    costeB <- valor máximo de entero

    vectorIntermedio <- nuevo arreglo de enteros de tamaño tamanioMatriz

    // Generar vector aleatorio inicial
    vectorIntermedio <- generarVectorAleatorio(tamanioMatriz, random)

    instanciaES <- ES(matrizFlujo, matrizDistancia, tamanioMatriz, random, vectorIntermedio)

    // Aplicar Enfriamiento simulado (ES) al vector aleatorio inicial
    mejorSolucion <- instanciaES.ES()

    Para i = 0 hasta 23:
        // Mutación
        vectorIntermedio <- mutarVector(mejorSolucion)
        instanciaES <- ES(matrizFlujo, matrizDistancia, tamanioMatriz, random, vectorIntermedio)
        vectorIntermedio <- instanciaES.ES()

        costeA = calcularCosteSolucion(matrizFlujo, matrizDistancia, vectorIntermedio)

        Si costeA < costeB entonces:
            this.mejorSolucion <- vectorIntermedio
            costeB <- costeA
        Fin si
    Fin para

    devolver mejorSolucion
Fin funcion
```

## 3.6. Búsqueda con Vecindario Variable

El algoritmo VNS consistirá en generar una solución inicial aleatoria y aplicar el algoritmo de BL sobre ella. A diferencia del ILS, cada vez que se aplique la BL se aplicará con un vecindario distinto. Si tras aplicar la BL se produce una mejora, vuelve al vecindario distinto. Cuando se haya aplicado el último vecindario volverá

al primero (aplicamos la variante circular). Una vez obtenida la solución optimizada, se estudiará si es mejor que la mejor solución encontrada hasta el momento y se realizará una mutación sobre la mejor de estas dos, volviendo a aplicar el algoritmo de BL sobre esta solución mutada. Este proceso se repite un determinado número de veces, devolviéndose la mejor solución encontrada en todo el proceso. Por tanto, se sigue el criterio del mejor como criterio de aceptación.

El vecindario determina el número de atributos que se cambia simultáneamente en la BL. Inicialmente k=1 (es decir, la BL pedida en esta clase), luego pasa a k = 2 (en este caso no se intercambian dos localizaciones, si no cuatro), luego k=3, y así sucesivamente. Si la BL produce una mejora, k pasa automáticamente a 1. Si k supera kmax se inicia a k=1.

**Atributos de Instancia**

- Las respectivas matrices de **flujo** y **distancia** junto con su respectivo tamaño que tendrá el vector solución.
- Atributo de tipo **Random** esto para poder realizar ejecuciones con una semilla. Para este y los demás problemas la semilla **random** será **42**.
- **Instancia** de la clase de Búsqueda Local (la modificada) para poder hacer la llamada a **algoritmoBL().**
- **Instancia** de la clase **QAP** para poder hacer la llamada a **generarVectorAleatorio()** y **calcularCosteSolucion().**
- **mejorSolucion** vector que irá almacenando posibles soluciones.
- **Random**, atributo utilizado para fijar semillas y que las soluciones sean siempre las mismas siempre y cuando se elija la opción de semilla.

**Pseudocódigo**

```
Función VNS(): arreglo de enteros
    vectorIntermedio = nuevo arreglo de enteros de tamaño tamanioMatriz
    k <- 1
    costeA <- 0
    costeB <- valor máximo de entero
    kMax <- 5

    vectorIntermedio <- generarVectorAleatorio(tamanioMatriz, random)

    BLm <- BL_Modificado(matrizFlujo, matrizDistancia, tamanioMatriz, random, vectorIntermedio, 2000)

    // Aplicar Búsqueda Local (BL) con k = 1 al vector aleatorio inicial
    mejorSolucion <- BLm.algoritmoBL(k)

    Para i = 0 hasta 23:
        vectorIntermedio <- mutarVector(mejorSolucion)

        BLm <- BL_Modificado(matrizFlujo, matrizDistancia, tamanioMatriz, random, vectorIntermedio, 2000)

        // Aplicar Búsqueda Local (BL) con k al vector mutado
        vectorIntermedio <- BLm.algoritmoBL(k)

        costeA <- calcularCosteSolucion(matrizFlujo, matrizDistancia, vectorIntermedio)

        si costeA < costeB entonces:
            k <- 1
            mejorSolucion <- vectorIntermedio
            costeB <- costeA
        sino:

            k <- k + 1

        si k > kMax entonces:
            k <- 1
   Fin para

    devolver mejorSolucion
Fin Funcion
```

# 4. Concepto del desarrollo de la Práctica

## 4.1. Lenguaje del Proyecto

La práctica 3 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
- **PairGreedy.java:** Para el inicio del proyecto, simulando que el vector solución era una dupla.
- **PairGenetico.java:** Para la práctica 2 poder representar de mejor manera los hijos de un cruce de dos padres.
- **Genetico.java:** Se desarrollan tanto el **Algoritmo Genético Generacional** como el **Algoritmo Genético Estacionario** ambos con los dos cruces posibles.
- **Memético.java:** Contiene copia de la clase **búsqueda local** y copia del **Algoritmo Genético Generacional** usando el cruce PMX y sus respectivos métodos de apoyo
- **ES.java:** Para la práctica 3, se realiza el algoritmo **ES**.
- **ILS.java:** Para la práctica 3, se realiza el algoritmo **ILS** y la mutación cíclica.
- **BMB.java:** Para la práctica 3, se realiza el algoritmo **BMB**.
- **ILS_ES.java:** Para la práctica 3, se realiza la hibridación entre el algoritmo **ILS** haciendo llamadas a **ES.**
- **VNS.java:** Para la práctica 3, se realiza el algoritmo **VNS** junto con la mutación cíclica.
- **BL_Modificado.java:** Para la práctica 3, se modifica la selección de las posiciones a intercambiar a que sean de forma aleatoria. Se crea el mismo método pero con capacidad de recibir un argumento para la llamada que se hace desde **VNS.**

## 4.2. 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
```

## 4.3. Ejecución del Proyecto

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

En la anterior práctica se me comentó que puedo compilar con Glade o Maven, pero honestamente no sé cómo hacerlo así que sigo haciéndolo manualmente para **Linux**

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

```
alumno@colladoalex:~/Escritorio/MH-P2/Software$ javac -d BIN FUENTES/qap/*.java
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
alumno@colladoalex:~/Escritorio/MH-P2/Software$
```

7. 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/Collado Rojas Alexander P3/Software/BIN$ java qa
p/QAP
0 - Algoritmo Greedy
1 - Algoritmo Busqueda Local
2 - Algoritmo Genetico
3 - Algoritmo Memetico
4 - Algoritmo Busqueda Local Modificado
5 - Algoritmo Busqueda Multiarranque Basica
6 - Algoritmo ILS
7 - Algoritmo VNS
8 - Algoritmo de Enfriamiento Simulado
9 - Algoritmo ILS ES

Elige cual algoritmo quieres usar:
8
0 - Semilla
1 - Aleatoria
Cual opcion quieres elegir?
0

ARCHIVO: tai100b.dat
COSTE: 1227273184
Tiempo de ejecucion: 80 milisegundos
```

### 7.1.1. Curiosidades de Ejecución

Al asignar semilla **Random** la ejecución en Netbeans difiere con la ejecución en Linux. Siempre darán los mismos resultados si se ejecuta con semilla en Netbeans y en Linux, pero entre estos dan distintos.

Por ejemplo, en Netbeans con el algoritmo **ES** el archivo tai100b.dat da como fitness 1254138988 y siempre dará este valor si se elige la opción de semilla, pero en Linux el mismo archivo da fitness de 1227273184 siempre que se ejecute con semilla. Desconozco la razón exacta, pero supongo que será de fijar la semilla **random a 42** que entre distintos sistemas operativos cambiará

# 8. Experimento y análisis de los resultados

## Enfriamiento Simulado

| Algoritmo Enfriamiento Simulado - Con Semilla | | | |
|---|---|---|---|
| **Caso** | **Coste obtenido** | **Fitness** | **Tiempo** |
| Chr22a | 7214 | 17.19 | 3.00 |
| Chr22b | 7704 | 24.38 | 0.00 |
| Chr25a | 6002 | 58.11 | 2.00 |
| Esc128 | 74 | 15.63 | 126.00 |
| Had20 | 6938 | 0.23 | 0.00 |
| Lipa60b | 3027298 | 20.12 | 3.00 |
| Lipa80b | 9515645 | 22.56 | 5.00 |
| Nug28 | 5370 | 3.95 | 0.00 |
| Sko81 | 92584 | 1.74 | 17.00 |
| Sko90 | 117518 | 1.72 | 18.00 |
| Sko100a | 154970 | 1.95 | 24.00 |
| Sko100f | 152562 | 2.37 | 24.00 |
| Tai100a | 22056058 | 4.77 | 14.00 |
| Tai100b | 1254138988 | 5.75 | 36.00 |
| Tai150b | 515178065 | 3.26 | 85.00 |
| Tai256c | 45724458 | 2.16 | 530.00 |
| Tho40 | 251742 | 4.67 | 1.00 |
| Tho150 | 8396492 | 3.23 | 77.00 |
| Wil50 | 49748 | 1.91 | 3.00 |
| Wil100 | 275760 | 1.00 | 19.00 |

## Búsqueda Local Modificada

| Algoritmo Busqueda Local Modificado- Con Semilla | | | |
|---|---|---|---|
| **Caso** | **Coste obtenido** | **Fitness** | **Tiempo** |
| Chr22a | 6768 | 9.94 | 12.00 |
| Chr22b | 7232 | 16.76 | 7.00 |
| Chr25a | 6158 | 62.22 | 8.00 |
| Esc128 | 76 | 18.75 | 39.00 |
| Had20 | 7074 | 2.20 | 5.00 |
| Lipa60b | 2520135 | 0.00 | 14.00 |
| Lipa80b | 9445942 | 21.66 | 19.00 |
| Nug28 | 5376 | 4.07 | 7.00 |
| Sko81 | 92758 | 1.93 | 20.00 |
| Sko90 | 118300 | 2.39 | 21.00 |
| Sko100a | 155496 | 2.30 | 25.00 |
| Sko100f | 151658 | 1.76 | 26.00 |
| Tai100a | 21800116 | 3.55 | 24.00 |
| Tai100b | 1215051551 | 2.45 | 26.00 |
| Tai150b | 519260575 | 4.08 | 52.00 |
| Tai256c | 45344562 | 1.31 | 101.00 |
| Tho40 | 250350 | 4.09 | 9.00 |
| Tho150 | 8291540 | 1.94 | 44.00 |
| Wil50 | 50014 | 2.45 | 12.00 |
| Wil100 | 275828 | 1.02 | 26.00 |

## Búsqueda Multiarranque Básica

| Algoritmo Busqueda Multiarranque Basical - Con Semilla | | | |
|---|---|---|---|
| **Caso** | **Coste obtenido** | **Fitness** | **Tiempo** |
| Chr22a | 6412 | 4.16 | 13.00 |
| Chr22b | 6570 | 6.07 | 5.00 |
| Chr25a | 5140 | 35.41 | 4.00 |
| Esc128 | 64 | 0.00 | 362.00 |
| Had20 | 6930 | 0.12 | 2.00 |
| Lipa60b | 3016659 | 19.70 | 45.00 |
| Lipa80b | 9419913 | 21.33 | 109.00 |
| Nug28 | 5272 | 2.05 | 5.00 |
| Sko81 | 92526 | 1.68 | 192.00 |
| Sko90 | 117048 | 1.31 | 269.00 |
| Sko100a | 154160 | 1.42 | 396.00 |
| Sko100f | 151018 | 1.33 | 386.00 |
| Tai100a | 21816956 | 3.63 | 224.00 |
| Tai100b | 1200667000 | 1.24 | 455.00 |
| Tai150b | 508163530 | 1.86 | 1911.00 |
| Tai256c | 44898158 | 0.31 | 4130.00 |
| Tho40 | 248444 | 3.30 | 18.00 |
| Tho150 | 8303966 | 2.10 | 1713.00 |
| Wil50 | 49084 | 0.55 | 39.00 |
| Wil100 | 275568 | 0.93 | 415.00 |

## Búsqueda Local Reiterada

| Algoritmo Busqueda Local Reiterada - Con | | | |
|---|---|---|---|
| **Caso** | **Coste obtenido** | **Fitness** | **Tiempo** |
| Chr22a | 6496 | 5.52 | 15.00 |
| Chr22b | 6642 | 7.23 | 7.00 |
| Chr25a | 4508 | 18.76 | 8.00 |
| Esc128 | 66 | 3.13 | 42.00 |
| Had20 | 6922 | 0.00 | 5.00 |
| Lipa60b | 3041652 | 20.69 | 16.00 |
| Lipa80b | 9535365 | 22.82 | 25.00 |
| Nug28 | 5286 | 2.32 | 7.00 |
| Sko81 | 93682 | 2.95 | 28.00 |
| Sko90 | 118936 | 2.94 | 36.00 |
| Sko100a | 155754 | 2.47 | 39.00 |
| Sko100f | 153392 | 2.92 | 40.00 |
| Tai100a | 22250342 | 5.69 | 36.00 |
| Tai100b | 1227090633 | 3.46 | 48.00 |
| Tai150b | 526490286 | 5.53 | 105.00 |
| Tai256c | 45243898 | 1.08 | 136.00 |
| Tho40 | 244296 | 1.57 | 11.00 |
| Tho150 | 8548446 | 5.10 | 99.00 |
| Wil50 | 49208 | 0.80 | 13.00 |
| Wil100 | 276450 | 1.25 | 44.00 |

## Hibridación ILS con ES

| Algoritmo ILS/ES - Con Semilla | | | |
|---|---|---|---|
| **Caso** | **Coste obtenido** | **Fitness** | **Tiempo** |
| Chr22a | 6552 | 6.43 | 24.00 |
| Chr22b | 6630 | 7.04 | 12.00 |
| Chr25a | 4774 | 25.76 | 15.00 |
| Esc128 | 66 | 3.13 | 3365.00 |
| Had20 | 6926 | 0.06 | 10.00 |
| Lipa60b | 3005738 | 19.27 | 79.00 |
| Lipa80b | 9408577 | 21.18 | 199.00 |
| Nug28 | 5292 | 2.44 | 15.00 |
| Sko81 | 92344 | 1.48 | 352.00 |
| Sko90 | 116920 | 1.20 | 505.00 |
| Sko100a | 153714 | 1.13 | 627.00 |
| Sko100f | 150976 | 1.30 | 591.00 |
| Tai100a | 21764446 | 3.38 | 420.00 |
| Tai100b | 1202632886 | 1.40 | 767.00 |
| Tai150b | 509245115 | 2.07 | 2072.00 |
| Tai256c | 45159734 | 0.89 | 13083.00 |
| Tho40 | 242842 | 0.97 | 42.00 |
| Tho150 | 8274678 | 1.74 | 1993.00 |
| Wil50 | 49122 | 0.63 | 84.00 |
| Wil100 | 274940 | 0.70 | 780.00 |

## Búsqueda con Vecindario Variable

| Algoritmo VNS - Con Semilla | | | |
|---|---|---|---|
| **Caso** | **Coste obtenido** | **Fitness** | **Tiempo** |
| Chr22a | 6352 | 3.18 | 23.00 |
| Chr22b | 6898 | 11.37 | 18.00 |
| Chr25a | 4448 | 17.18 | 20.00 |
| Esc128 | 66 | 3.13 | 69.00 |
| Had20 | 6926 | 0.06 | 11.00 |
| Lipa60b | 3014685 | 19.62 | 32.00 |
| Lipa80b | 9437175 | 21.55 | 52.00 |
| Nug28 | 5248 | 1.59 | 16.00 |
| Sko81 | 93096 | 2.31 | 45.00 |
| Sko90 | 116810 | 1.10 | 63.00 |
| Sko100a | 153926 | 1.27 | 75.00 |
| Sko100f | 151196 | 1.45 | 67.00 |
| Tai100a | 21934030 | 4.19 | 64.00 |
| Tai100b | 1200031733 | 1.18 | 67.00 |
| Tai150b | 514683614 | 3.16 | 165.00 |
| Tai256c | 45137394 | 0.84 | 204.00 |
| Tho40 | 243768 | 1.35 | 21.00 |
| Tho150 | 8353328 | 2.70 | 127.00 |
| Wil50 | 49224 | 0.84 | 30.00 |
| Wil100 | 275388 | 0.86 | 85.00 |

## Valores Medios

| Algoritmo | Media Fitness | Media Tiempo (ms) |
|---|---|---|
| BL P1 | 7.74 | 26.35 |
| BL modificada | 8.24 | 24.85 |
| AM All | 5.08 | 1071.4 |
| BMB | 5.42 | 534.65 |
| ILS | 5.81 | 38 |
| ES | 9.83 | 49.35 |
| ILS-ES | 5.11 | 1251.75 |
| VNS | 4.95 | 62.7 |

## Análisis de los Resultados

Se pueden observar en las tablas que las **búsquedas basadas en trayectorias** nos dan mejores resultados que las anteriores prácticas para el problema de la asignación cuadrática.

Para la **búsqueda multiarranque básica** es curioso ver de como nos mejora bastante la desviación ejecutando la búsqueda local de la práctica 1 varias veces y quedándonos con una mejor solución cada vez que se ejecuta con la única desventaja de que el tiempo medio de ejecución sea un poco más alto. Esta mejora hace que este algoritmo bastante simple nos de uno de los mejores resultados para esta práctica.

Para el **enfriamiento simulado**, es interesante ver el poco tiempo de ejecución para ser uno de los más complicados de implementar. Esto se debe a que con el método **Cauchy** el enfriamiento es muy rápido haciendo pocas iteraciones y dando el peor de los resultados de esta práctica. Creo que si se realizan más iteraciones poniendo la temperatura final con un menor valor o tal vez haciendo otro proceso de enfriamiento de tal forma que **T** vaya enfriando más lento.

Al igual que el **BMB** el algoritmo **ILS** es un algoritmo muy sencillo de implementar que da resultados bastante buenos y además de esto con un tiempo muy rápido. Donde solamente varía la mutación que se le realiza al vector solución buscando mejores resultados en cada una de sus iteraciones. Y bueno además de esto también se modifica la búsqueda local evitando el **Don´t look bits.**

El **VNS** es el que da los mejores resultados esto podría ser porque explora más soluciones en el vecindario de la búsqueda local respecto a los **k** pares a intercambiar.

Por último, creo que la hibridación de **ILS** con **ES** da mejores resultados que el **ILS** normal ya que al no utilizar búsqueda local la solución puede explorar distintas soluciones mejores y no estancarse en óptimos locales.