UNIVERSIDAD DE GRANADA

Metaheurística

Memoria de la práctica 2 de Metaheurística con el problema de asignación cuadrática (QAP)

Realizado por:
Alexander Collado Rojas

colladoalex@correo.ugr.es

Mayo 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.

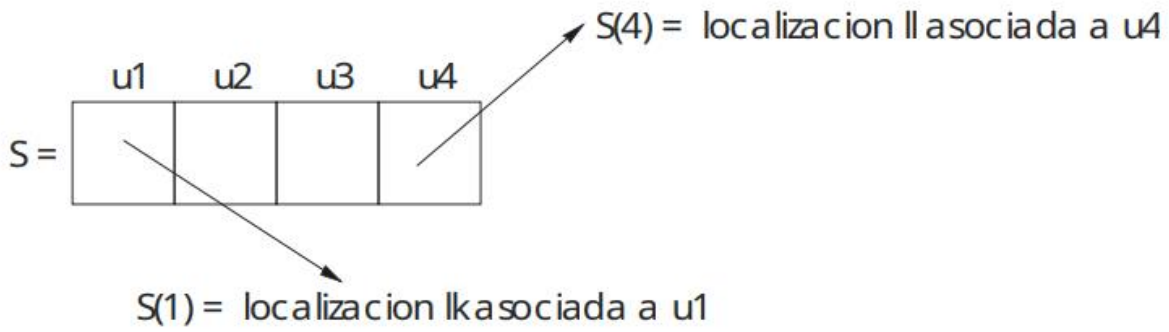
El tamaño y las matrices serán presentadas en un documento **.dat** que tiene la estructura de:

- 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
```

2.6. Clase PairGenetico.java

Esta clase se ha creado para poder representar la salida de los **operadores de cruce**, esto ya que tanto en **crucePosicion** como en **crucePMX** la salida eran dos hijos que su representación sería una permutación de tamaño **tamanioMatriz**. Es decir, pair genético es una estructura para poder representar dos hijos en una sola salida.

La razón de por la cual se tuve que implementar esto fue porque tenía un problema con java que no me dejaba utilizar la clase Pair que se encuentra en la librería de Java **javafx.util.Pair.**

3. Metaheurística Basada en Poblaciones

Una metaheurística basada en poblaciones es una técnica de optimización que utiliza un conjunto de soluciones potenciales (población) para explorar y buscar soluciones óptimas en un espacio de búsqueda. Estas metaheurísticas se inspiran en conceptos biológicos y emplean operadores evolutivos para guiar la búsqueda hacia regiones prometedoras del espacio de soluciones.

3.1. Algoritmo Genético

El algoritmo genético generacional elitista simula la evolución biológica utilizando conceptos como la selección natural, los cruces y la mutación. Inicialmente, se crea una población de soluciones aleatorias llamadas "individuos". Luego, se evalúa la aptitud de cada individuo en función a su respectivo coste. Los individuos con menor coste tienen más probabilidades de sobrevivir en la siguiente población, la supervivencia depende del modelo que se vaya a seguir. A través de operadores genéticos como el cruce y la mutación, se generan nuevas soluciones combinando características de los individuos existentes. Este proceso se repite durante varias generaciones, de tal manera que se busca que la población mejore en sus costes.

Los algoritmos genéticos serán representados en una clase **Genetico.java** en la cual tendremos como atributos de instancias:

- **Matrices de Flujo y Distancia**
- **Poblacion:** La población consistirá de 50 individuos generados aleatoriamente, es decir 50 permutaciones de **tamanioMatriz**. Se representa mediante una matriz en la cual cada fila es un individuo.
- **Constantes;**
    - **tamanioPoblacion:** Indica el tamaño de la población, es decir cuantas filas contendrá la matriz.
    - **probabilidadCruceAGG, probabilidadCruceAGE,** indican cada una de las probabilidades de que tienen los individuos de una población para cruzar. En el **AGG** de una población de 50 individuos hay 35 individuos que pueden cruzarse, pero como el cruce se hace entre dos padres se ha redondeado hacia arriba para que sean 36 individuos los que crucen.

    - **probabilidadMutacion** es las probabilidades que tienen los genes en mutar. Por ejemplo, en una población de 50 individuos con 14 genes la probabilidad de mutación sería de 7 genes.
- **Random rand:** Se utiliza por si se quiere ejecutar el algoritmo utilizando una semilla, de tal manera que el algoritmo con la misma semilla siempre da los mismos resultados.
- **Evaluacion:** Contabilizará cada vez que se evalúe un individuo, es decir cada vez que se ejecute la función de **calcularlCoste**.

A continuación, se describirán los **operadores y funciones comunes** que se han utilizado en ambos modelos:

Operador de Selección

Se usará el torneo binario, consistente en elegir aleatoriamente dos individuos de la población y seleccionar el mejor de ellos. Para esto se utilizará el método **selección** cuyos parámetros será la población de la cual se desee seleccionar y la cantidad de padres que se quieran generar. Devolverá una nueva población de tamaño **cantidadDePadres** que será la población de la cual se realizarán todos los cruces.

```
Función seleccion(poblacionParametro: matriz de enteros, cantidadPadres: entero) -> matriz de enteros
    costeInd1: entero
    costeInd2: entero

    padres: matriz de enteros
    padres: nueva matriz de enteros de tamaño [cantidadPadres][tamanioMatriz]

    contadorPadre <- 0

    ind1 <- 0
    ind2 <- 0

    Para i = 0 hasta cantidadPadres exclusivo hacer
        ind1 <- obtenerEnteroAleatorio(0, tamanioPoblacion)
        ind2 <- obtenerEnteroAleatorio(0, tamanioPoblacion)

        costeInd1 <- calcularCoste(poblacionParametro, ind1)
        costeInd2 <- calcularCoste(poblacionParametro, ind2)

        Si costeInd1 < costeInd2 entonces
            padres[contadorPadre] <- copiarFila(poblacionParametro, ind1)
            contadorPadre <- contadorPadre + 1
        Sino
            padres[contadorPadre] <- copiarFila(poblacionParametro, ind2)
            contadorPadre <- contadorPadre + 1
        Fin Si
    Fin Para

    Devolver padres
Fin Función
```

Operador de Mutación

Para el operador de mutación será necesario generar dos números aleatorios, estos indicarán a cuál **individuo** se le realizará la mutación de la posición **gen**. Se realizarán solo una cantidad fija de mutaciones, indicadas al inicio de cada algoritmo (generacional o estacionario). El operador de mutación solamente consiste en intercambiar la posición de este **gen** respecto a otra.

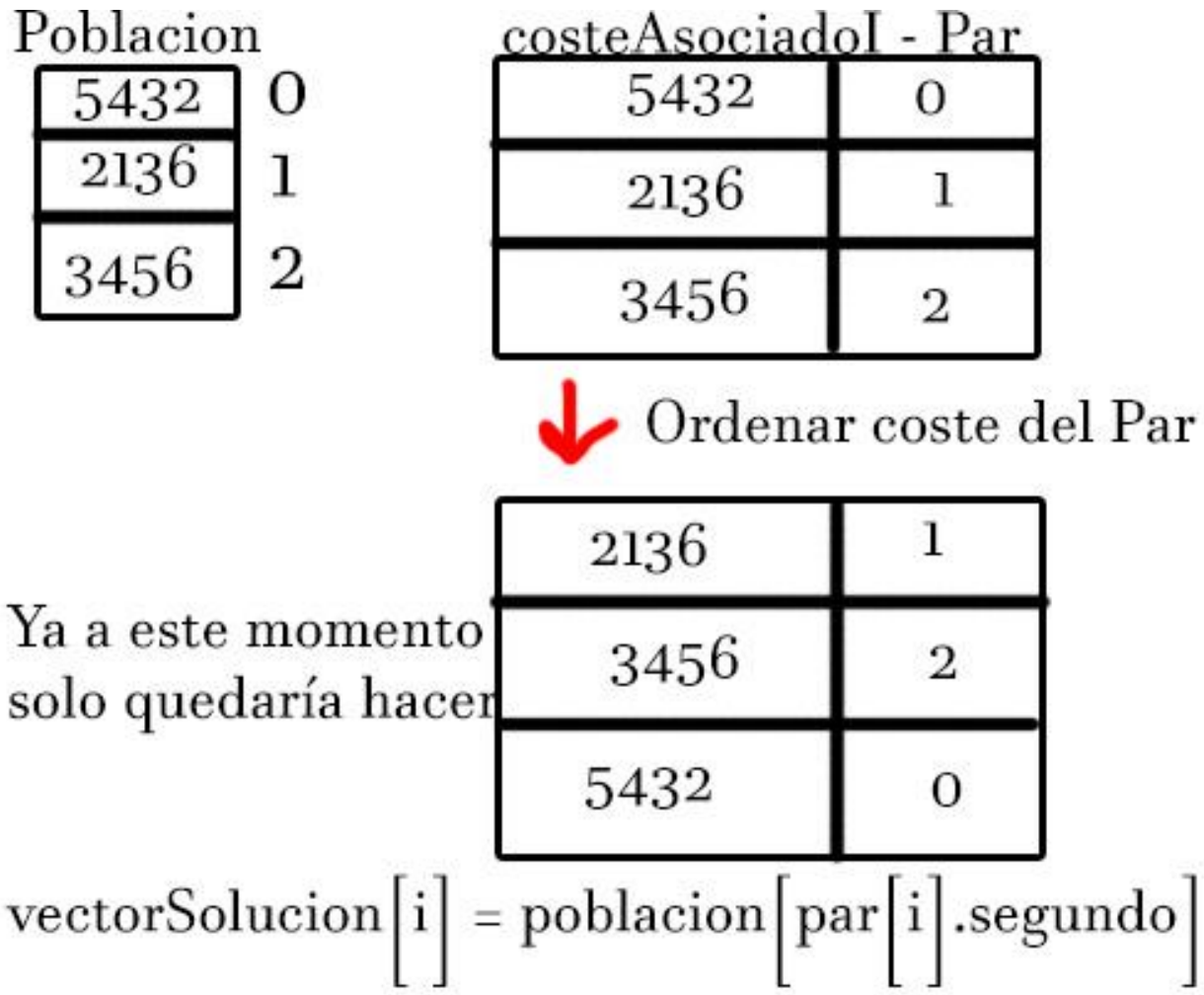
```
Función mutacion(genAMutar: entero, individuoAMutar: arreglo de enteros) -> arreglo de enteros
    posicionPrimera: entero <- genAMutar
    posicionSegunda: entero <- -1
    intercambioPosicionPrimera: entero
    salir: booleano <- falso

    Mientras no salir hacer
        intercambioPosicionPrimera <- obtenerEnteroAleatorio(0, tamanioMatriz)
        posicionSegunda <- buscarPosicion(individuoAMutar, intercambioPosicionPrimera)

        Si no (posicionPrimera == posicionSegunda) entonces
            salir = verdadero
        Fin Si
    Fin Mientras

    aux: entero = individuoAMutar[posicionPrimera]
    individuoAMutar[posicionPrimera] <- individuoAMutar[posicionSegunda]
    individuoAMutar[posicionSegunda] <- aux

    Devolver individuoAMutar
Fin Función
```

A este operador se le ha añadido una función extra de apoyo **buscarPosicion** que consiste en buscar la posición en la que se encuentra el **gen** con el que se realizará el intercambio. También existe una comprobación para que la posición a intercambiar no sea igual a la del **gen**.

Función ordenarPoblacion

Esta función tomará la población inicial y la ordenará de menor a mayor coste. Para ordenar la población se tomará un **par de enteros** los cuales consisten en el coste del individuo y su posición en el vector original de población. Luego se ordenará el primer elemento del par, es decir se ordenarán los **costes.** Luego se creará un vector auxiliar para asignarle la población en la posición i respecto al **par**.

La siguiente imagen ilustra lo que se comenta:



La función ordenar tiene otra función de apoyo para poder ordenar por el método de burbuja el par mencionado. El pseudocódigo de la función entera sería:

```
//Se devuelve la población aunque no es necesario ya que poblacion es un atributo de instancia
Función ordenarPoblacion() -> matriz de enteros
    costeAsociadoI: arreglo de PairGreedy de tamaño tamanioPoblacion
    vectorAuxiliar: matriz de enteros de tamaño [tamanioPoblacion][tamanioMatriz]

    Para i = 0 hasta tamanioPoblacion exclusivo hacer
        costeAsociadoI[i] <- nuevo PairGreedy(calcularCoste(poblacion, i), i)
    Fin Para

    bubbleSort(costeAsociadoI)

    Para i = 0 hasta tamanioPoblacion exclusivo hacer
        vectorAuxiliar[i] <- copiarFila(poblacion, costeAsociadoI[i].getSegundo())
    Fin Para

    poblacion <- vectorAuxiliar

    Devolver poblacion
Fin Función
```

Operador de Cruce Posición

El operador de cruce posición consistirá en elegir dos padres, los genes que tengan la misma posición en ambos padres se mantendrán en sus respectivos hijos mientras que los demás genes se reestructurarán de manera aleatoria en el individuo.

```
Función crucePosicion(primerPadre: arreglo de enteros, segundoPadre: arreglo de enteros) -> PairGenetico<arreglo de enteros, arreglo de enteros>
    hijo: arreglo de enteros de tamaño tamanioMatriz
    elegido: arreglo de booleanos de tamaño tamanioMatriz
    contador: entero <- 0

      //Un vector utilizado para comparar genes en posiciones
    Para i = 0 hasta tamanioMatriz exclusivo hacer
        elegido[i] <- falso
    Fin Para

      //Si el gen del primer padre coincide con la misma posicion en el segundo padre este pasara al hijo
    Para i = 0 hasta tamanioMatriz exclusivo hacer
        Si primerPadre[i] == segundoPadre[i] entonces
            hijo[i] <- primerPadre[i]
            elegido[i] <- verdadero
            contador <- contador + 1
        Fin Si
    Fin Para

      //Necesario para que los hijos sean diferentes, sino en Java me da error
    hijoaux: arreglo de enteros <- copiarArreglo(hijo)
```

```
      //Lista con los elementos que no coinciden en posicion
     listaPermutar: lista de enteros
     Para i = 0 hasta tamanioMatriz exclusivo hacer
         Si no elegido[i] entonces
             listaPermutar.agregar(primerPadre[i])
         Fin Si
     Fin Para

     hijo1: arreglo de enteros <- generacionHijoPosicion(elegido, hijo, listaPermutar)
     hijo2: arreglo de enteros <- generacionHijoPosicion(elegido, hijoaux, listaPermutar)

     parHijos: PairGenetico<arreglo de enteros, arreglo de enteros> <- nuevo PairGenetico(hijo1, hijo2)

     Devolver parHijos
Fin Función
```

El método generacionHijoPosicion toma aquellos valores que no han sido añadidos al hijo y los asigna de manera aleatoria, teniendo en cuenta que la posición no haya sido ya elegida anteriormente. La posición elegida se puede implementar con un vector booleano que indica si la posición **i** contiene el mismo gen en ambos padres.

Operador de Cruce PMX

El operador de cruce PMX consistirá nuevamente en elegir ambos padres, se asignará una franja aleatoria en la cual los genes de esa franja del primer padre pasarán al segundo hijo y los del segundo padre pasarán al primer hijo. Los demás genes se ordenarán respecto a las correspondencias entre los genes de ambos padres, si uno de estos genes de la correspondencia ya existiera dentro de los

nuevos genes del hijo, esta correspondencia se volverá a aplicar hasta que el valor del gen no esté repetido en el hijo. En la siguiente imagen se puede observar como se aplica este cruce.

p1 = { 2, 4, 6, 7, 1, 5, 3, 0}

p2 = {1, 3, 7, 2, 4, 0, 5, 6}

A este punto cada hijo i se rellenara mediante la correspondencia del elemento en la posicion j de cada padre. Es decir para el hijo 1 el primer elemento sera la correspondencia del valor 2

h1 = {*, *, *, 2, 4, 0, 5, *}

↕ ↕ ↕ ↕

h2 = { *, *, *, 7, 1, 5, 3, *}

correspondencias

Las correspondencias para h1 se leen de h1 a h2. Las correspondencias para h2 se leen de h2 a h1.

Por ejemplo, para el hijo1

- Primera posicion
  - El 2 corresponde con 7, el 7 al no ser un valor utilizado en el vector este se puede colocar
- Segunda posicion
  - El 4 corresponde con el 1, se coloca
- Tercera posicion
  - El 6 no es una correspondencia asi que se mantiene el valor
- Cuarta - Septima posicion
  - Los valores de padre2
- Octava posicion
  - El 0 corresponde con el 5, al estar ya el 5 dentro del vector se buscará la correspondencia de este
    - El 5 corresponde con el 3, se coloca en el vector

h1 = {7, 1, 6, 2, 4, 0, 5, 3}

h2 = { 4, 0, 2, 7, 1, 5, 3, 6}

```
Función crucePMX(primerPadre: arreglo de enteros, segundoPadre: arreglo de enteros) -> PairGenetico<arreglo de enteros, arreglo de enteros>
    primerHijo: arreglo de enteros de tamaño tamanioMatriz
    segundoHijo: arreglo de enteros de tamaño tamanioMatriz
    posicionInicial: entero <- 0
    posicionFinal: entero <- 0

    Mientras no (posicionInicial < posicionFinal) hacer
        posicionInicial <- obtenerEnteroAleatorio(0, tamanioMatriz)
        posicionFinal <- obtenerEnteroAleatorio(posicionInicial, tamanioMatriz)
    Fin Mientras

    Para i = posicionInicial hasta posicionFinal + 1 hacer
        primerHijo[i] <- segundoPadre[i]
        segundoHijo[i] <- primerPadre[i]
    Fin Para

    correspondencia: arreglo de PairGreedy de tamaño (posicionFinal - posicionInicial) + 1
    contadorCorrespondencia: entero <- 0

      //Pair Greedy es una estructura para agrupar dos enteros
    Para i = 0 hasta tamanioMatriz hacer
        Si i >= posicionInicial y i <= posicionFinal entonces
            correspondencia[contadorCorrespondencia] <- nuevo PairGreedy(segundoPadre[i], primerPadre[i])
            contadorCorrespondencia <- contadorCorrespondencia + 1
        Fin Si
    Fin Para
```

```
    primerHijo <- auxiliarOperadorPMX(correspondencia, primerHijo, primerPadre, posicionInicial, posicionFinal, falso)
    segundoHijo <- auxiliarOperadorPMX(correspondencia, segundoHijo, segundoPadre, posicionInicial, posicionFinal, verdadero)

    parHijos: PairGenetico<arreglo de enteros, arreglo de enteros> <- nuevo PairGenetico(primerHijo, segundoHijo)

    Devolver parHijos
Fin Función
```

El método llamado **auxiliarOperadorPMX** es el encargado de realizar todo el tema de las correspondencias, de tal forma que este recibirá el vector de **pares de correspondencias**, el **primer hijo** con el **primer padre** del cual se basarán estas correspondencias, la **franja de las posiciones** que no se deben modificar y un **valor** que indicará verdadero cuando la correspondencia se lea de h2 a h1 y falso cuando la correspondencia se lea de h1 a h2.

En este método se utiliza un vector de enteros de auxiliar que se rellenará en -1 aquellas posiciones que no sean la franja, la idea de esto es que cuando se busque si el valor está repetido no haga la comparación con el 0(que es uno de los valores incluidos en la permutación). El pseudocódigo es el siguiente:

```
Función auxiliarOperadorPMX(correspondencia: arreglo de PairGreedy, hijo: arreglo de enteros, padre: arreglo de enteros, posicionInicial: entero, posicionFinal: entero,
eleccion: booleano) -> arreglo de enteros
    hijoResult: arreglo de enteros de tamaño tamanioMatriz
    elegido: booleano
    valorEnCorrespondencia: entero
    auxiliarIndex: entero <- 0

    Para i = 0 hasta tamanioMatriz hacer
        Si i >= posicionInicial y i <= posicionFinal entonces
            hijoResult[i] <- hijo[i]
        Sino
            hijoResult[i] <- -1
        Fin Si
    Fin Para

      //Metodo de lectura de correspondencia
            //Para hijo 1 correspondencia primera -> segunda
            //Para hijo 2 correspondencia segunda -> primera
    Si eleccion es verdadero entonces
        correspondencia = intercambiarParCorrespondencia(correspondencia)
    Fin Si

    Para i = 0 hasta tamanioMatriz hacer
        elegido <- falso
        valorEnCorrespondencia <- padre[i]
```

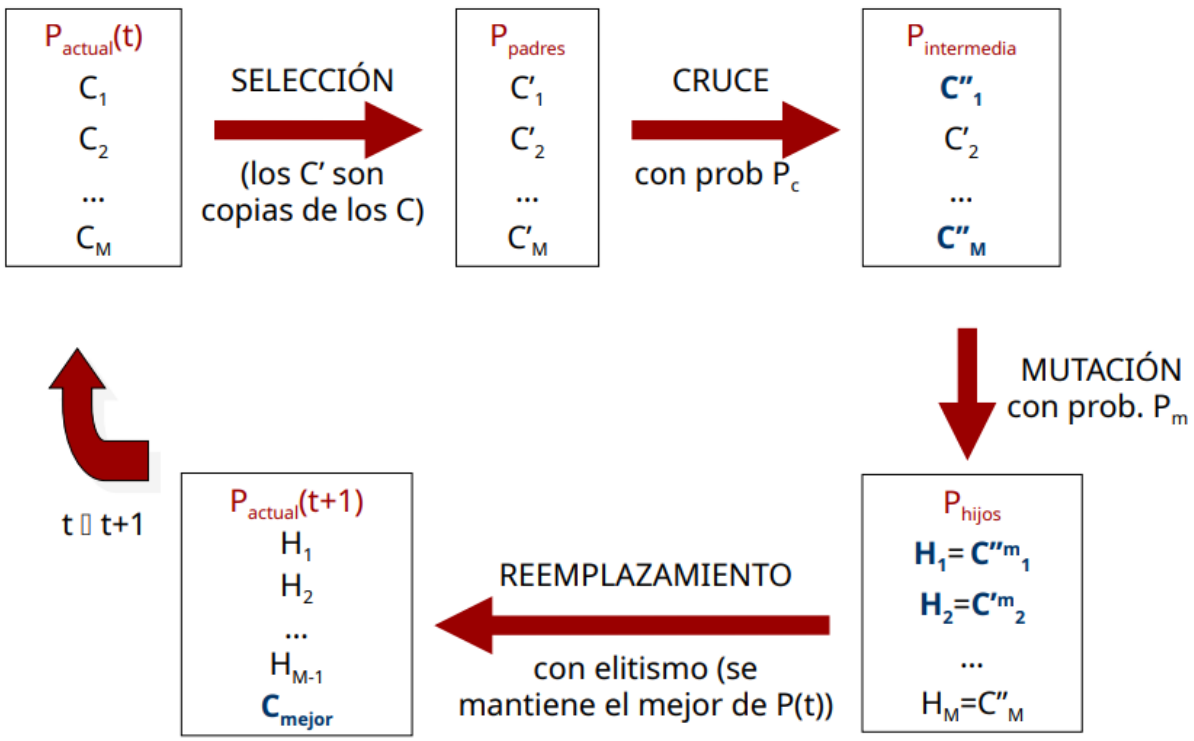
```
            //Si no se encuentra en la franja
        Si i < posicionInicial o i > posicionFinal entonces
            Mientras no elegido hacer
                auxiliarIndex <- posicionEnCorrespondencia(correspondencia, valorEnCorrespondencia) //Indica la posicion en el vector de par correspondencia

                Si auxiliarIndex es distinto de -1 entonces                                  // Significa que existe esta correspondencia
                    valorEnCorrespondencia <- correspondencia[posicionEnCorrespondencia(correspondencia, valorEnCorrespondencia)].getSegundo()

                    Si no valorRepetido(hijoResult, valorEnCorrespondencia) entonces
                        hijoResult[i] <- valorEnCorrespondencia
                        elegido <- verdadero
                    Fin Si
                Sino
                    hijoResult[i] <- valorEnCorrespondencia
                    elegido <- verdadero
                Fin Si
            Fin Mientras
        Fin Si
    Fin Para

    Devolver hijoResult
Fin Función
```

Dentro de este método existen otros tales como:

- **intercambiarParCorrespondencia**
  - Es utilizado para poder leer las correspondencias de una manera u de otra. El primer elemento de la correspondencia pasa a ser el segundo elemento y viceversa.
- **valorRepetido**
  - Comprobará si el valor que se busque insertar en el vector ya exista. Por este método fue necesario hacer que las posiciones no definidas se pusieran a -1.
- **posicionEnCorrespondencia**
  - Comprobará si el elemento que se busca existe en el vector de correspondencia (en el primer elemento del par). Si existe devuelve su posición, si no existe devuelve -1.

3.1.1. Algoritmo Genético Generacional

El algoritmo genético generacional elitista consiste en:



- En el operador **selección** se elegirán 50 padres de la población inicial mediante el torneo binario.
- El esquema de reemplazo consistirá en que los 50 nuevos hijos generados a partir de los padres, reemplazarán a la población actual, exceptuando el hijo cuyo tenga peor coste, este hijo será reemplazado por el padre que tenga el mejor coste.

```
Funcion reemplazarPoblacionAGG(poblacionHijos)
    ordenarPoblacion()

    mejorPadre <- poblacion[0]
    costeMejorPadre <- calcularCoste(poblacion, 0)

      //Hijos reemplazan a la poblacion actual
    poblacion <- poblacionHijos

    ordenarPoblacion()
    costePeorHijo <- calcularCoste(poblacion, tamanioPoblacion - 1)

    Si costeMejorPadre < costePeorHijo then
        poblacion[tamanioPoblacion - 1] <- mejorPadre
    Fin Si
Fin Funcion
```

- El cruce tendrá una probabilidad del 70% de todos los padres, como se ha comentado anteriormente se cruzarán 36 padres generando 36 hijos. Los restantes 14 hijos serán idénticos a los padres no cruzados.

El pseudocódigo del AGG es el siguiente

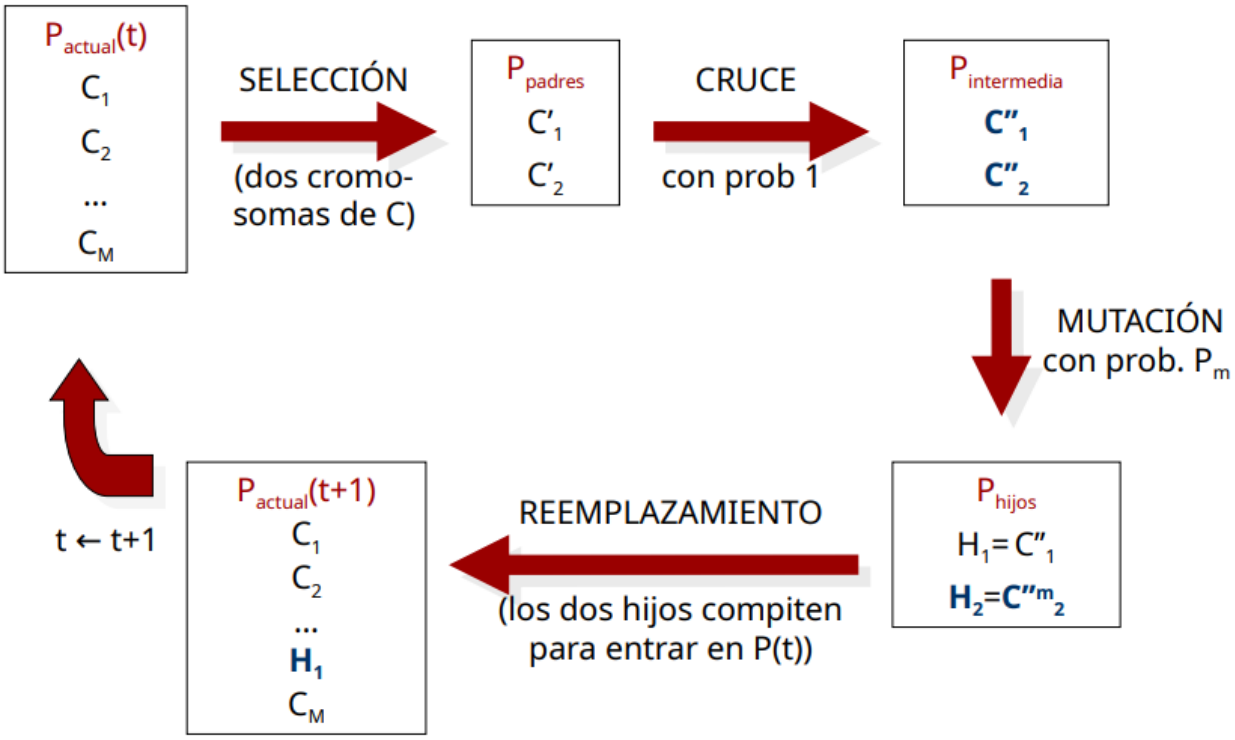
```
Función AGG() -> arreglo de enteros
    padres: matriz de enteros
    hijos: matriz de enteros de tamaño tamanioPoblacion
    parHijos: arreglo de PairGenetico de tamaño tamanioPoblacion / 2

    cruce: flotante <- probabilidadCruceAGG * (tamanioPoblacion / 2)
    cruceEsperado: entero <- redondear(cruce)

    mutacion: flotante <- (tamanioPoblacion * tamanioMatriz) * probabilidadMutacionGenes
    mutacionEsperada: entero <- redondear(mutacion)

    ordenarPoblacion()

    Mientras evaluacion < iter_max hacer

        padres <- seleccion(poblacion, tamanioPoblacion)


        contador = 0

        Para j = 0 hasta (cruceEsperado * 2) - 1 hacer
            parHijos[contador] <- cruce(padres[j], padres[j+1])
            hijos[j] <- parHijos[contador].getPrimero()
            hijos[j+1] <- parHijos[contador].getSegundo()
            contador++
        Fin Para

        Para k = cruceEsperado * 2 hasta tamanioPoblacion hacer
            hijos[k] <- padres[k]
        Fin Para
```

```
        gen <- 0
        individuo <- 0

        Para j = 0 hasta mutacionEsperada hacer

            gen <- valor random limite tamanioMatriz
            individuo <- valor random limite tamanioPoblacion

            hijos[individuo] <- mutacion(gen, hijos[individuo])

        Fin Para


        reemplazarPoblacionAGG(hijos)
        generacion++
    Fin Mientras

    ordenarPoblacion()
    Devolver poblacion[0]   //Devuelve el mejor individuo de toda la poblacion
Fin Función
```

Aunque en el código se hayan dividido en **AGG_Posicion** y **AGG_PMX** ambos usarán el mismo código solamente cambia el operador de cruce llamando a su correspondiente cruce.

3.1.2. Algoritmo Genético Estacionario

El algoritmo genético estacionario consiste en:



- En el operador **selección** se elegirán 2 padres de la población inicial mediante el torneo binario.
- El esquema de reemplazo consistirá en que los 2 nuevos hijos generados a partir de los padres competirán con los 2 peores padres para entrar en la siguiente generación.

```
Funcion reemplazarPoblacionAGE(hijos)
    ordenarPoblacion()

    posicion1 <- -1
    posicion2 <- -1

    candidatos: lista de arreglos de enteros <- [hijos[0], hijos[1], poblacion[tamanioPoblacion - 1], poblacion[tamanioPoblacion - 2]]

    Para i = 0 hasta candidatos.tamaño -1 hacer
        Si posicion1 == -1 o calcularCoste(candidatos[i]) < calcularCoste(candidatos[posicion1]) hacer
            posicion2 <- posicion1
            posicion1 <- i
        Sino Si posicion2 == -1 o calcularCoste(candidatos[i]) < calcularCoste(candidatos[posicion2]) hacer
            posicion2 <- i
        Fin Si
    Fin Para

    poblacion[tamanioPoblacion - 2] <- candidatos[posicion1]
    poblacion[tamanioPoblacion - 1] <- candidatos[posicion2]

    ordenarPoblacion()
Fin Funcion
```

- El cruce tendrá una probabilidad del 100% de todos los padres, esto hace que los dos padres que han sido seleccionados cruzarán, de forma que generarán los dos nuevos hijos.

El pseudocódigo del AGE es el siguiente:

```
Función AGE() -> arreglo de enteros
    padres: matriz de enteros
    hijos: matriz de enteros de tamaño tamanioAGE //Solo se generan dos nuevos hijos

      //Como se generan dos hijos solo hace falta un parGenetico
    parHijos: arreglo de PairGenetico de tamaño tamanioAGE / 2

    cruce: flotante <- probabilidadCruceAGE * (tamanioPoblacion / 2)
    cruceEsperado: entero <- redondear(cruce)

    mutacion: flotante <- (tamanioAGE * tamanioMatriz) * probabilidadMutacionGenes
    mutacionEsperada: entero <- redondear(mutacion)

    ordenarPoblacion()

    Mientras evaluacion < iter_max hacer

        padres = seleccion(poblacion, tamanioAGE) //TamanioAGE es igual a 2


        contador <- 0

        Para j = 0 hasta cruceEsperado  hacer
            parHijos[contador] <- cruce(padres[j], padres[j+1])
            hijos[j] <- parHijos[contador].getPrimero()
            hijos[j+1] <- parHijos[contador].getSegundo()
            contador++
        Fin Para

        gen <- 0
        individuo <- 0
```

```
        gen <- 0
        individuo <- 0

        Para j = 0 hasta mutacionEsperada hacer
            gen <- valor entero random
            individuo <- valor entero random
            hijos[individuo] <- mutacion(gen, hijos[individuo])
        Fin Para


        reemplazarPoblacionAGE(hijos)
        generacion++
    Fin Mientras

    ordenarPoblacion()
    Devolver poblacion[0]
Fin Función
```

Al igual que en el AGG el cruce dependerá de cual se elija

4. Algoritmos Meméticos

Algoritmo basado en la evolución de poblaciones que para realizar búsqueda heurística intenta utilizar todo el conocimiento sobre el problema, para el caso de esta práctica se ha utilizado el algoritmo genético generacional con cruce PMX junto con el algoritmo de búsqueda local.

El pseudocódigo será el mismo utilizado en el AGG, exceptuando que al final se tendrá una comprobación de que si la generación que se está ejecutando es múltiplo de 10 entonces se aplica el Algoritmo Memético seleccionado. Este código se incluye luego de hacer el reemplazo.

```
Si generacion % 10 == 0 hacer
    Selecionar(memetico){
        caso 0:
            AM_All();
            break;
        caso 1:
            AM_Rand();
            break;
        caso 2:
            AM_Best();
            break;
    }
}
```

Existe un atributo de instancia adicional **memetico** que indicará cual es el algoritmo que se desee ejecutar, este atributo será indicado en el constructor.

Para la implementación de esta clase se copió el **algoritmo genético con cruce PMX** y sus respectivos métodos de apoyo y la clase entera de **búsqueda local**. La idea principal era hacerlo por llamadas a funciones de otra clase, pero como como se deben crear instancias me habría dado problemas con respecto a la evaluación.

Además, cada vez que se realice una **factorización** de la **Búsqueda Local** se sumará como una evaluación.

4.1. Aplicados a toda la población

Luego de reemplazar la población de inicial por la población de hijos se ejecutará la **búsqueda local** con máximo de 400 exploraciones a vecinos o encontrar la mejor solución a **cada uno** de los individuos de la población. El resultado de esta **búsqueda local** reemplazará al individuo de la población.

```
Función AM_All()

    Para i = 0 hasta tamanioPoblacion hacer
        poblacion[i] <- algoritmoBL(poblacion[i])
        costeSolucion <- 0


        Para j = 0 hasta tamanioMatriz hacer
            vectorBinario[j] <- falso
        Fin Para
    Fin Para

Fin Función
```

4.2. Aplicados a N individuos de manera aleatoria

Luego de reemplazar a la población inicial por la población de hijos se realizará la **búsqueda local** a **N** individuos de la población elegidos de manera aleatoria. Como la población para estos problemas será de 50, la probabilidad de realizar el algoritmo es del 0.1, entonces se elegirán 5 individuos aleatoriamente.

```
Función AM_Rand()
    seleccion <- tamanioPoblacion * pLS
    seleccionEsperada <- redondear(seleccion)

    Para i = 0 hasta seleccionEsperada hacer
        cromosomaElegido <- generarEnteroAleatorio(0, tamanioPoblacion)

        poblacion[cromosomaElegido] <- algoritmoBL(poblacion[cromosomaElegido])
        costeSolucion <- 0

        Para j = 0 hasta tamanioMatriz hacer
            vectorBinario[j] <- falso
        Fin Para
    Fin Para
Fin Función
```

4.3. Aplicados a los N mejores individuos

Luego de hacer el reemplazo de la población inicial por la población de hijos se aplicará el algoritmo de **búsqueda local** a los 5 mejores individuos de la población.

```
Función AM_Best()
    seleccion = tamanioPoblacion * pLS
    seleccionEsperada = redondear(seleccion)

    Para i = 0 hasta seleccionEsperada hacer
        poblacion[i] = algoritmoBL(poblacion[i])
        costeSolucion = 0

        Para j = 0 hasta tamanioMatriz hacer
            vectorBinario[j] = falso
        Fin Para
    Fin Para
Fin Función
```

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
- **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

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 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 como hacerlo así que sigo haciéndolo manualmente para **Linux**

- Asegurarse de tener el kit de desarrollo (JDK) instalado
- 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$
```

- 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-P2/Software$ cd BIN
alumno@colladoalex:~/Escritorio/MH-P2/Software/BIN$ java qap/QAP
0 - Algoritmo Greedy
1 - Algoritmo Busqueda Local
2 - Algoritmo Genetico
3 - Algoritmo Memetico

Elige cual algoritmo quieres usar:
2
0 - Semilla con BL
1 - Aleatoria
Cual opcion quieres elegir?
0
0 - AGG Posicion
1 - AGG PMX
2 - AGE Posicion
3 - AGE PMX
Cual opcion quieres elegir?
0

ARCHIVO: tai100b.dat
COSTE: 1625505748
Tiempo de ejecucion: 481 milisegundos
```

5.3.1. Novedades de Ejecución

En la práctica anterior se debía hacer una a una la ejecución de cada archivo introduciéndole el nombre, para esta práctica solo hace falta elegir cual algoritmo utilizar y se mostrará por pantalla el resultado de cada uno de los algoritmos junto con sus tiempos. El único problema es que los resultados no salen en el mismo orden, sino que por ejemplo puede salir primero el **tai100b.dat** antes que el **chr22a.dat**.

Tanto los genéticos como los meméticos se pueden ejecutar con una semilla, esta semilla será el vector resultante de realizar una búsqueda local a una ejecución greedy. Este vector será permutado generando una población inicial de 50. Como se indica semilla en el random, cada vez que se permute dará los mismos resultados.

6. Experimento y resultados

Para evaluar los algoritmos implementados tanto el **Genético** como los **Meméticos** 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 1000000 para convertirlos en milisegundos que será lo que salga por pantalla y lo indicado en las tablas

Aclaración: Los tiempos han sido medidos ejecutando el programa en Netbeans.

1. **Genético Generacional Cruce Posicion**

Algoritmo Genetico Generacional Posicion-Con Semilla			
Caso	**Coste obtenido**	**Fitness**	**Tiempo**
Chr22a	8874	44.15	72.00
Chr22b	10366	67.36	33.00
Chr25a	13240	248.79	35.00

Esc128	236	268.75	656.00
Had20	7180	3.73	22.00
Lipa60b	3223742	27.92	147.00
Lipa80b	9997327	28.77	247.00
Nug28	6284	21.64	36.00
Sko81	105742	16.20	276.00
Sko90	133330	15.40	307.00
Sko100a	174530	14.82	377.00
Sko100f	170516	14.41	363.00
Tai100a	23706442	12.61	371.00
Tai100b	1625505748	37.06	361.00
Tai150b	627142802	25.71	826.00
Tai256c	50896380	13.71	2343.00
Tho40	308698	28.35	63.00
Tho150	9550252	17.42	809.00
Wil50	54076	10.78	98.00
Wil100	296360	8.54	359.00

2. Genético Generacional Cruce PMX

Algoritmo Genetico Generacional PMX- Con Semilla			
Caso	**Coste obtenido**	**Fitness**	**Tiempo**
Chr22a	6524	5.98	76.00
Chr22b	6934	11.95	25.00
Chr25a	5756	51.63	29.00
Esc128	78	21.88	609.00
Had20	6972	0.72	21.00
Lipa60b	3063385	21.56	140.00
Lipa80b	9619798	23.90	261.00
Nug28	5518	6.81	365.00
Sko81	96026	5.53	257.00
Sko90	123994	7.32	314.00

Sko100a	162946	7.20	365.00
Sko100f	157602	5.75	370.00
Tai100a	22723500	7.94	379.00
Tai100b	1350163060	13.84	380.00
Tai150b	569081721	14.07	842.00
Tai256c	47425754	5.96	2460.00
Tho40	257162	6.92	64.00
Tho150	9082906	11.67	841.00
Wil50	50400	3.24	56.47
Wil100	282788	3.57	93.00

3. Genético Estacionario Cruce Posicion

Algoritmo Genetico Estacionario Posicion- Con Semilla			
Caso	**Coste obtenido**	**Fitness**	**Tiempo**
Chr22a	9974	62.02	37.00
Chr22b	9870	59.35	25.00
Chr25a	11170	194.26	29.00
Esc128	238	271.88	557.00
Had20	7368	6.44	19.00
Lipa60b	3215043	27.57	130.00
Lipa80b	9964719	28.35	226.00
Nug28	6042	16.96	30.00
Sko81	103790	14.06	252.00
Sko90	132692	14.85	276.00
Sko100a	171306	12.70	327.00
Sko100f	170242	14.23	339.00
Tai100a	23503240	11.64	329.00
Tai100b	1529003354	28.92	348.00
Tai150b	616473145	23.57	755.00
Tai256c	49273678	10.09	2182.00
Tho40	296156	23.13	54.00
Tho150	9488282	16.66	747.00
Wil50	53066	8.71	83.00

Wil100	292972	7.30	331.00

4. Genético Estacionario Cruce PMX

Algoritmo Genetico Estacionario PMX- Con Semilla			
Caso	**Coste obtenido**	**Fitness**	**Tiempo**
Chr22a	8910	44.74	35.00
Chr22b	9402	51.79	29.00
Chr25a	9052	138.46	27.00
Esc128	160	150.00	565.00
Had20	7074	2.20	18.00
Lipa60b	3156975	25.27	124.00
Lipa80b	9852054	26.89	229.00
Nug28	5914	14.48	32.00
Sko81	101148	11.15	231.00
Sko90	130264	12.75	275.00
Sko100a	170180	11.96	335.00
Sko100f	165276	10.90	341.00
Tai100a	23277822	10.57	363.00
Tai100b	1474166096	24.30	352.00
Tai150b	602644748	20.80	868.00
Tai256c	48195862	7.68	2325.00
Tho40	285190	18.57	55.00
Tho150	9350504	14.96	820.00
Wil50	52184	6.90	88.00
Wil100	289730	6.11	335.00

5. Memético All

Algoritmo Memetico All

Caso	Coste obtenido	Fitness	Tiempo
Chr22a	6478	5.23	22.00
Chr22b	6480	4.62	12.00
Chr25a	5012	32.03	16.00
Esc128	64	0.00	717.00
Had20	6922	0.00	8.00
Lipa60b	3008616	19.38	95.00
Lipa80b	9417016	21.29	223.00
Nug28	5272	2.05	12.00
Sko81	92142	1.26	407.00
Sko90	117050	1.31	519.00
Sko100a	154102	1.38	824.00
Sko100f	151172	1.43	776.00
Tai100a	21732280	3.23	468.00
Tai100b	1202119683	1.36	1039.00
Tai150b	508482441	1.92	3923.00
Tai256c	44874002	0.26	8282.00
Tho40	244106	1.49	35.00
Tho150	8283504	1.85	3212.00
Wil50	49258	0.91	73.00
Wil100	274636	0.59	765.00

6. Memético Random

Algoritmo Memetico Random			
Caso	**Coste obtenido**	**Fitness**	**Tiempo**
Chr22a	6722	9.19	25.00
Chr22b	6788	9.59	23.00
Chr25a	5642	48.63	16.00
Esc128	68	6.25	106.00
Had20	6926	0.06	14.00

Lipa60b	3018790	19.79	32.00
Lipa80b	9440824	21.60	32.00
Nug28	5284	2.28	16.00
Sko81	92890	2.08	48.00
Sko90	117454	1.66	69.00
Sko100a	154468	1.62	91.00
Sko100f	151934	1.94	88.00
Tai100a	21732280	3.23	61.00
Tai100b	1207965107	1.85	87.00
Tai150b	511190076	2.46	427.00
Tai256c	44961198	0.45	839.00
Tho40	243366	1.18	18.00
Tho150	8288462	1.91	336.00
Wil50	49142	0.67	19.00
Wil100	275328	0.84	88.00

7. Memético Best

Algoritmo Memetico			
Caso	**Coste obtenido**	**Fitness**	**Tiempo**
Chr22a	6732	9.36	28.00
Chr22b	6836	10.36	22.00
Chr25a	5816	53.21	17.00
Esc128	68	6.25	95.00
Had20	6972	0.72	18.00
Lipa60b	2996097	18.89	31.00
Lipa80b	9425686	21.40	31.00
Nug28	5318	2.94	17.00
Sko81	92648	1.81	49.00
Sko90	117730	1.90	65.00
Sko100a	154518	1.66	96.00
Sko100f	151918	1.93	86.00
Tai100a	21799640	3.55	56.00
Tai100b	1204428673	1.55	102.00

Tai150b	510345786	2.29	419.00
Tai256c	44922456	0.36	886.00
Tho40	251104	4.40	18.00
Tho150	8306722	2.13	352.00
Wil50	49242	0.87	20.00
Wil100	275992	1.08	80.00

Valores Medios:

Algoritmo	Media Fitness	Media Tiempo
Greedy	54.98	0.01308
BL	7.74	0.08167
AGG Posicion	46.31	390.5
AGG PMX	11.87	397.37
AGE Posicion	42.63	353.8
AGE PMX	30.52	372.35
AM All	5.08	1071.4
AM Random	6.86	121.75
AM Best	7.33	124.4

Los tiempos han sido medidos en milisegundos

Análisis de los Resultados

Como resultado de las ejecuciones de los algoritmos se pueden concluir que:

- Entre todos los algoritmos genéticos el que mejor converge hacia una solución óptima sería el **AGG PMX** pero esto por varias razones:
    - La primera creo que sería el proceso de **selección** de los padres, en este proceso por torneo binario solo se elegirán padres determinados por los que tengan un mejor coste, a partir de estas selecciones se irán creando los hijos.
    - La segunda razón viene a ser el operador de cruce, como se cruzan los hijos respecto a franjas aleatorias muchas de las veces estas franjas tienen un coste bastante óptimo y ya que los hijos heredan estas franjas en cada generación que pasa irá convergiendo la solución a una más global.
    - Por último, el esquema de reemplazo indica que los hijos serán la nueva población a excepción del último. Esto quiere decir que en cada

generación que va pasando se irá descartando la peor de las soluciones y mejorando las demás.

- El algoritmo que peor converge sería el **AGG Posicion,** la razón de esto es que para el operador de cruce la presencia de aleatoriedad es mucho mayor y muchas veces dependerá de la población inicial a la cual se le busca aplicar al algoritmo. Por ejemplo, en una permutación de 22 elementos muchas veces ninguna de las permutaciones coincide y esto haría que ambos hijos sean simplemente una nueva permutación de sus padres, pudiendo así tener hasta un peor coste.
- En los estacionarios el **AGE PMX** converge peor que el **AGG PMX** pero algo mejor que el **AGE Posicion** la razón es:
    - El proceso de selección solo selecciona **dos padres** de manera aleatoria de la población inicial, que serán los dos que mejor coste tengan de los 4 buscados.
    - Estos dos padres realizan sus operadores de cruces correspondiente
        - Para el caso de Posición pasa igual que el caso de **AGG Posicion** la presencia de la aleatoriedad puede influir negativamente en las soluciones.
        - Para el caso de PMX acontece igual, las franjas guardan un coste que muchas veces es correcto pasándolo así a los nuevos hijos.
    - La parte **importante** es en el esquema de reemplazo **AGE**:
        - Ambos hijos competirán con el peor de los padres para entrar en la población, de esta manera se podrán ir generando por generación nuevos individuo hijos con mejores costes. Parece que la convergencia de este esquema de reemplazo será más lenta pero más segura, de forma que a mayor número de generaciones mejor serán nuestros resultados.
- Para los algoritmos meméticos el que mejor resultado nos indica es el **AM All**, la razón principal es que se les aplica la búsqueda local a los 50 individuos de la nueva generación, esta búsqueda local en diferentes permutaciones podrá ir encontrando mejores locales y reemplazándolo por el individuo, luego solo tocará reordenar el resultado de la nueva población para así encontrar una mejor opción. Se tienen 50 mejores locales y alguno de estos se irá acercando al mejor global. El único problema de este algoritmo es que al tener que realizar **búsqueda local** a todos los individuos estos deben refactorizar muchas veces haciendo que solo se ejecuten 10

generaciones. Otra desventaja es que realizar tantas búsquedas locales influye en los tiempos de ejecución dependiendo del tamaño de los algoritmos.

- Los otros dos **algoritmos meméticos** funcionan de una manera parecida, solo que la cantidad de individuos a los que se la aplica la **búsqueda local** es mucho menor, por tanto, los tiempos son menores. En el caso con mi semilla el **AM Random** me ha dado mejores resultados que el **AM Best** esto ya que las permutaciones elegidas aleatoriamente tenían un **mejor local** mejores que las elegidas de primera. El ejemplo lo muestro en la siguiente ilustración