INFORME PRÁCTICA 2

Metaheurísticas

Grupo 7:

David Rodríguez Muro, 26523873C
drm00035@red.ujaen.es
Juan Bautista Muñoz Ruiz, 26516720C
jbmr0001@eed.ujaen.es

Algoritmos implementados: Algoritmo Genético Estacional, con cruces PMX y OX y Algoritmo Genético Generacional, con Cruces PMX y OX2

Grupo de prácticas 4, viernes (10:30-12:30)
Curso 2021/2022

ÍNDICE

DESCRIPCIÓN DEL PROBLEMA Y METAHEURÍSTICA UTILIZADA

PROBLEMA A RESOLVER

Nos encontramos ante el mismo problema que en la práctica anterior pero usando los datos de NISSAN además de los de FORD.

Los ingenieros de estas multinacionales necesitan realizar un estudio de su fábrica para mejorar el flujo de piezas entre los distintos departamentos o unidades de fabricación.

El objetivo de este problema es encontrar la fórmula de distribución de los departamentos, conociendo la distancia entre los distintos departamentos y el flujo de piezas que existe.

Como vimos en el seminario se trata de un problema combinatorio de gran complejidad *NP-completo* de elevado coste computacional. Además la eficiencia de este problema se podría considerar cuadrática o incluso cúbica en algunos problemas similares. En cuanto a la estructura de los datos:

- Encontramos ***p*** departamentos donde **(p_i, i=1, ..., n)**
- Encontramos ***I*** localizaciones donde **(l_j , j=1, ... ,n)**
- Se definen dos matrices **F=(f_{ij})** y **D=(d_{ij})** con una dimensión cuadrada de n con la siguiente información:
    - **F** es la matriz de flujo de piezas, es decir, **f_{ij}** es el número de piezas que pasan del departamento i al departamento j.
    - **D** es la matriz de distancias, es decir, **d_{ij}** es la distancia entre el departamento i y el departamento j.
    - El coste de asignar p_i a l_k y p_j a l_l es:

      $$f_{ij} * d_{kl} + f_{ji} * d_{lk}$$

Matemáticamente el problema se centra en minimizar el coste de las asignaciones de los departamentos con las distintas localizaciones

$$min \quad \sum_{i,j=1}^{n} \sum_{k,p=1}^{n} f_{ij} d_{kp} x_{ij} x_{kp}$$

$$s.a. \quad \sum_{i=1}^{n} x_{ij} = 1 \quad 1 \leq j \leq n,$$
$$\sum_{j=1}^{n} x_{ij} = 1 \quad 1 \leq i \leq n,$$
$$x_{ij} \in \{0,1\} \quad 1 \leq i \leq n$$

REPRESENTACIÓN DE LA SOLUCIÓN

Representaremos la solución mediante un vector de enteros:

Descripción del vector:

- **N** se corresponde con el número total de unidades de fabricación o departamentos.
- Cada índice de la posición del vector se corresponde con una unidad de fabricación o departamento.
- En cada posición del vector almacenaremos un entero con la unidad de fabricación o departamento asignada en función de nuestros algoritmos.

Restricciones del vector:

- No debe haber elementos repetidos.
- Todas las unidades de fabricación índice deben tener un entero con una unidad de fabricación asignada en su posición.
- Los enteros asignados a cada índice deben encontrarse entre cero y el tamaño total (número de unidades de fabricación)

EVALUACIÓN DE LA SOLUCIÓN

En la evaluación de la solución utilizamos la siguiente función de factorización:

$$\Delta(\phi, r, s) = \begin{bmatrix} f_{rr} \cdot (d_{\pi_s\pi_s} - d_{\pi_r\pi_r}) + \\ f_{ss} \cdot (d_{\pi_r\pi_r} - d_{\pi_s\pi_s}) + \\ f_{rs} \cdot (d_{\pi_s\pi_r} - d_{\pi_r\pi_s}) + \\ f_{sr} \cdot (d_{\pi_r\pi_s} - d_{\pi_s\pi_r}) + \end{bmatrix} + \sum_{k=1, k\neq r,s}^{n} \begin{bmatrix} f_{kr} \cdot (d_{\pi_k\pi_s} - d_{\pi_k\pi_r}) + f_{ks} \cdot (d_{\pi_k\pi_r} - d_{\pi_k\pi_s}) + \\ f_{rk} \cdot (d_{\pi_s\pi_k} - d_{\pi_r\pi_k}) + f_{sk} \cdot (d_{\pi_r\pi_k} - d_{\pi_s\pi_k}) + \end{bmatrix}$$

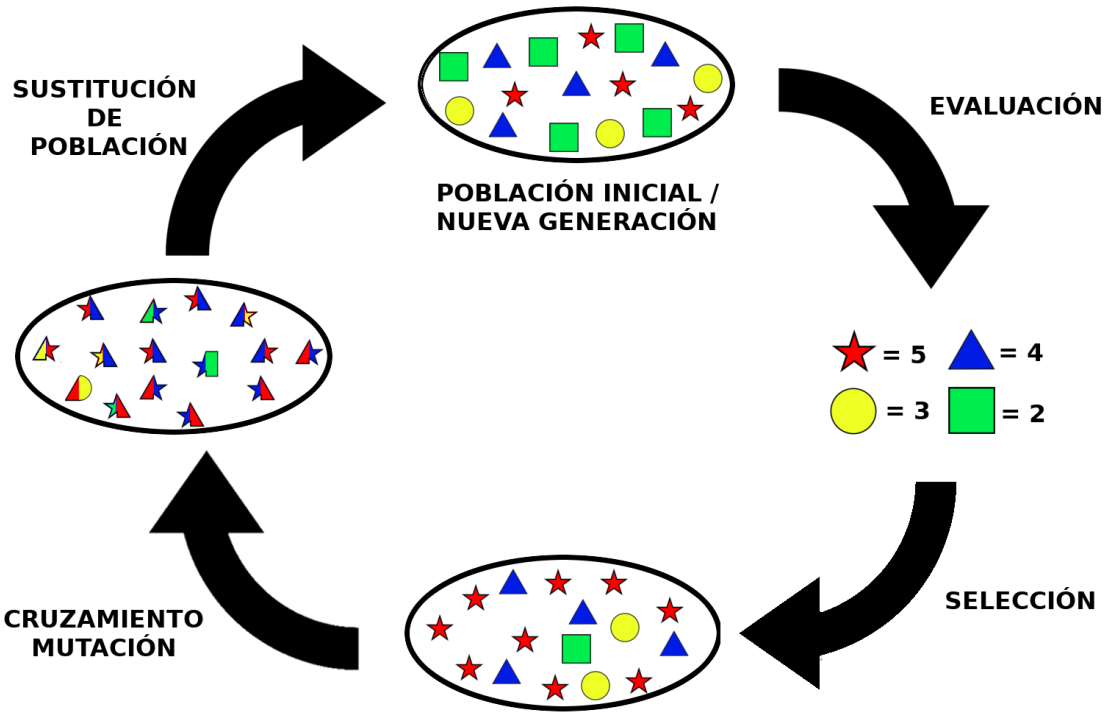
Gracias a esta fórmula podemos observar si el movimiento de intercambio a realizar mejora respecto a la solución anterior, ya que en caso de que el valor de la función de factorización sea negativo nuestra nueva solución sería mejor que la anterior. Al encontrarnos con un problema de optimización la solución sería aceptada, en caso contrario se descarta.

```
Funcion factorizacion

COMIENZO

   valor += CalculaFuncionl()
   PARA k = 0 HASTA k < tamTotal HACER
      SI r ! k && s != k ENTONCES
         valor += Funcion(k, r, s)
       TERMINASI
    TERMINAPARA
FIN
```

EXPLORACIÓN DEL ENTORNO

Exploramos el entorno mediante los algoritmos genéticos. Algoritmos fundados en los modelos de evolución basados en poblaciones, en concreto en los principios de las ideas de la evolución de Darwin.

Efectuaremos la exploración siguiendo un modelo evolutivo en el cual crearemos descendencia a partir de una población inicial. Población de la cual elegiremos unos padres mediante un proceso de selección diferente para cada algoritmo.



A partir de estos padres se generará descendencia aplicando unos operadores de cruce y de mutación que posteriormente serán explicados. Se culminará el proceso con la sustitución de la población ya sea de forma parcial o completa según los criterios que explicaremos en siguientes puntos para nuestro Algoritmo Genético Estacionario o Genético Generacional.

Estos algoritmos resultan ser una técnica de optimización probabilística que con frecuencia mejora a otros métodos clásicos en problemas difíciles. En el análisis final comprobaremos si mejoran los algoritmos de la primera práctica entregada.

MOVIMIENTO POR EL ENTORNO

Nos moveremos por el entorno a través de los operadores de cruce y mutación.

En líneas generales, en primer lugar, aplicaremos un cruce entre dos vectores de solución dados. Seguidamente, las soluciones recombinadas obtenidas las someteremos a una mutación para algunos de sus genes de manera aleatoria.

Con la aplicación de estos operadores iremos alterando los antecesores seleccionados para generar nuevas soluciones hijo, soluciones que tras ser evaluadas puede que mejoren nuestro óptimo actual.

Aunque serán explicados en profundidad en el apartado Operadores:

- Cruce: Combinamos un vector con elementos de otro siguiendo diferentes métodos o criterios estudiados como puede ser el OX, OX2 y PMX.
- Mutación: Aplicamos un operador 2-opt entre los genes elegidos aleatoriamente para mutar. En caso de quedar un gen calificado como “mutable” que no disponga de ninguno con el que permutar no se mutará.

OPERADORES

OPERADOR DE MUTACIÓN

Desde un nivel más alto iremos invocando permutaciones de dos en dos entre las posiciones del vector que hayan sido calificadas como “mutables”.

En el nivel más bajo realizaremos la mutación con el operador de permutación ya implementado en la práctica 1, un Operador 2-opt mediante el cual permutamos dos posiciones (i,j) indicadas.

De la forma que una permutación **(2,5)** en el vector solución:
[1, 7, 9, 6, 10, 11, 12, 5, 0, 3, 13, 17, 8, 19, 4, 18, 15, 14, 16, 2]

Daría resultado al vector solución:
[1, **10**, 9, 6, **7**, 11, 12, 5, 0, 3, 13, 17, 8, 19, 4, 18, 15, 14, 16, 2]

A continuación podemos ver su implementación en pseudocódigo:

```
Funcion permutacion

BEGIN

   a = vector[i]
   b = vector[j]

   vector[i] = b
   vector[j] = a

END
```

OPERADOR DE CRUCE PMX

Contamos con vectores solución padres [1 2 3 4 5 6 7 8] y [3 7 5 1 6 8 2 4] en los que seleccionamos dos puntos de corte aleatorios. Por ejemplo, entre la posición cuatro y seis , ambas incluidas. Aquí podemos ver los vectores padre con el corte marcado:

Padre 1: [1 2 3 | 4 5 6 | 7 8]
Padre 2: [3 7 5 | 1 6 8 | 2 4]

Posteriormente realizamos un mapeo entre las subcadenas dentro del corte, es decir, a cada elemento dentro del corte del primero padre le asignamos elemento dentro del corte del segundo padre. En nuestro ejemplo, quedaría así el mapeo para las subcadenas | 4 5 6 | y | 1 6 8 |:

4 <-> 1
5 <-> 6
6 <-> 8

Ahora la sección de asignación del Padre 1 se copia en el Hijo 2, y la sección de asignación de Padre 2 es copiado en Hijo 1:

Padre 1: [1 2 3 | 4 5 6 | 7 8] Hijo 1: [*** 1 6 8 **]

Padre 2: [3 7 5 | 1 6 8 | 2 4] Hijo 2: [*** 4 5 6 **]

Seguidamente, la descendencia del hijo i se completa copiando los elementos del padre i. En caso de que un número ya esté presente en la descendencia, se reemplaza de acuerdo con las asignaciones del mapeo. En el caso del Hijo 1 completamos copiando los elementos del Padre 1.

		Mapeo
Padre 1: [1 2 3 | 4 5 6 | 7 8]	Hijo 1: [*** 1 6 8 **]	4 <-> 1
Padre 2: [3 7 5 | 1 6 8 | 2 4]	Hijo 2: [*** 4 5 6 **]	5 <-> 6
		6 <-> 8

Hijo 1:

- El primer elemento de la descendencia 1 sería un 1. Sin embargo, ya hay un 1 presente en la descendencia 1. Por lo tanto, debido al mapeo 1 <-> 4, elegimos el 4:

  Hijo 1: [**4**** 1 6 8 **]

- El segundo, tercero y séptimo elementos de la descendencia 1 se pueden tomar del primer padre:

  Hijo 1: [4 **2 3** 1 6 8 **7***]

- Sin embargo, el último elemento de la descendencia 1 sería un 8, pero ya está presente. Por lo que debido a las asignaciones 8 <-> 6 y 6 <-> 5, se elige un 5.

Por lo tanto:

Padre 1: [1 2 3 4 5 6 7 8] Hijo 1: [**4 2 3** 1 6 8 **7 5**]
Padre 2: [3 7 5 1 6 8 2 4] Hijo 2: [**3 7 8** 4 5 6 **2 1**]

En cuanto a la implementación del cruce:

- En primer lugar generamos dos puntos de corte aleatorios el primero entre 0 y tamTotal-3 y el segundo entre el primero punto de corte y tamTotal-2 para asegurarnos un corte que deje elementos tanto a su izquierda como a su derecha. Además guardaremos los padres en una Lista auxiliar para guardar el padre completo.
- Crearemos el Hijo 1 clonando el Padre 2 y el Hijo 2 clonando el padre 1. Seguidamente estableceremos a -1 todas las posiciones que se encuentren fuera del corte en estos hijos.

- A continuación en dos listas enlazadas guardaremos los elementos cortados de cada padre para utilizarlas en el posterior mapeo. Mapeo en el que usando un array de enteros mapeo, a cada elemento i de la Lista enlazada 1 le asignamos el elemento i de la Lista Enlazada corte 2.
- Para finalizar por cada hijo i iremos rellenando cada posición con su padre i, es decir el Hijo 1 con el Padre 1, etc...:
    - Si la posición está marcada a -1:
        - Si el hijo ya contiene elemento del padre:
            - En un bucle hasta que el hijo no contenga este elemento:
                - LLamamos a mapeo[elem].
            - Actualizamos la posición con el elemento obtenido en el mapeo.
        - Si el el hijo no contiene esa posición del padre:
            - Estableceremos en esa posición el elemento que se encuentra en la misma posición en el padre.

A continuación podemos ver el pseudocódigo del algoritmo:

```
Funcion crucePMX

COMIENZO

    x = aleatorio.Randint(0, tamTotal - 3)
    y = aleatorio.Randint(x, tamTotal - 2)

    realizarCorte(escogido1, x, y)
    realizarCorte(escogido2, x, y)

    PARA i = 0 HASTA tamTotal HACER
        SI (i < x || i > y) ENTONCES
            hijo1[i] = -1
            hijo2[i] = -1
        TERMINASI
    TERMINAPARA

    PARA i = x HASTA i <= y HACER
        escogido1[i] = auxPadre2[i]
        corte1.add(auxPadre1[i])
        escogido2[i] = auxPadre1[i]
        corte2.add(auxPadre2[i])
    TERMINAPARA

    PARA i = 0 HASTA tamTotal HACER
        mapeo[i] = 0
    TERMINAPARA

    PARA i = 0 HASTA corte1.size() HACER
        mapeo[corte2[i]] = corte1[i]
    TERMINAPARA
```

```
    PARA i = 0 HASTA auxPadre1.size() HACER
        SI hijo1[i] == -1 ENTONCES
            SI contiene(auxPadre1[i], hijo1) ENTONCES
                encontrado = falso
                elem = auxPadre1[i]
                MIENTRAS !encontrado HACER
                    SI !contiene(elem, hijo1) ENTONCES
                        encontrado = verdadero
                    SINO
                        elem = mapeo[elem]
                    TERMINASI
                TERMINAMIENTRAS
            SINO
                hijo1[i] = auxPadre1[i]
            TERMINASI
        TERMINASI
    TERMINAPARA

    PARA i = 0 HASTA corte2.size() HACER
        mapeo[corte1[i]] = corte2[i]
    TERMINAPARA

    PARA i = 0 HASTA auxPadre2.size() HACER
        SI hijo2[i] == -1 ENTONCES
            SI contiene(auxPadre2[i], hijo2) ENTONCES
                encontrado = falso
                elem = auxPadre2[i]
                MIENTRAS !encontrado HACER
                    SI !contiene(elem, hijo2) ENTONCES
                        encontrado = verdadero
                    SINO
                        elem = mapeo[elem]
                    TERMINASI
                TERMINAMIENTRAS
```

```
            SINO
                hijo2[i] = auxPadre2[i]
            TERMINASI
        TERMINASI
    TERMINAPARA

    escogido1 = hijo1
    escogido2 = hijo2

FIN
```

OPERADOR DE CRUCE OX

Partimos de los Padres: [1 2 3 4 5 6 7 8] y [2 4 6 8 7 5 3 1]. Suponemos un punto de corte de la posición 3 a la 5 ambas incluidas:

[1 2 | 3 4 5 | 6 7 8]
[2 4 | 6 8 7 | 5 3 1]

Para la descendencia partimos de la copia de los cortes en cada Padre:

Padre 1: [1 2 3 4 5 6 7 8] Hijo 1: [* * | 3 4 5 |* * *]
Padre 2: [2 4 6 8 7 5 3 1] Hijo 2: [* *| 6 8 7 | * * *]

A partir de la posición del segundo punto de corte, el resto de elementos se copian en el orden en que aparecen en el otro padre partiendo desde esa posición. Omitiendo los elementos que ya están presentes. Cuando se llega al final del vector volvemos a la posición cero para rellenar las posiciones anteriores al primer punto de corte:

Hijo 1: [**8 7** | 3 4 5 | **1 2 6**]
Hijo 2: [**4 5** | 6 8 7 | **1 2 3**]

Para la implementación de este cruce:

- Clonamos el vector 1 en *auxPadre2* y el vector 2 en *auxPadre2*.
- Llamamos a la función *realizarCorte* con los dos vectores recibidos, función que pone a 9999 las posiciones del hijo fuera del corte para que al buscarlas no se encuentren e indicar que esa posición es rellenable.
- En un bucle por posición desde el segundo punto de corte hasta que llegamos al primer punto de corte:
    - Si el elemento de *auxPadre2* en la posición actual no está contenido en *Hijo1*, lo insertamos e incrementamos la posición actual en el hijo.
    - Avanzamos +1 en la posición del padre para colocarnos en la siguiente.
    - Comprobamos si hemos llegado al final del vector en el padre o en el hijo para volver al inicio de estos y poder recorrer el vector completo, ya que nuestra posición inicial partía del segundo punto de corte.

Para el otro hijo seguiremos el mismo proceder. Pseudocódigo:

```
Funcion CruceOX

COMIENZO

    x = aleatorio.Ranint(1, tamTotal - 3)
    y = aleatorio.Ranint(x, tamTotal - 2)

    realizarCorte(esc1, x, y)
    realizarCorte(esc2, x, y)

    i = y + 1
    j = y + 1
    MIENTRAS i != x ENTONCES
        esta = contiene(auxPadre2[j], esc1)
        SI esta ENTONCES
            esc1[i] = auxPadre2[j]
            i += 1
        TERMINASI
        j += 1
        SI j == tamTotal ENTONCES
            j = 0
        TERMINASI

        SI i == tamTotal ENTONCES
            i = 0
        TERMINASI
    TERMINAMIENTRAS



    i = y + 1
    j = y + 1
    MIENTRAS i != x ENTONCES
        esta = contiene(auxPadre1[j], esc2)
        SI esta ENTONCES
            esc2[i] = auxPadre1[j]
            i += 1
        TERMINASI
        j += 1
        SI j == tamTotal ENTONCES
            j = 0
        TERMINASI

        SI i == tamTotal ENTONCES
            i = 0
        TERMINASI
    TERMINAMIENTRAS

FIN
```

OPERADOR DE CRUCE OX2

Considerando los padres: [1 2 3 4 5 6 7 8] y [2 4 6 8 7 5 3 1], elegimos varias posiciones de forma aleatoria en el segundo padre. Suponemos la segunda, tercera y sexta posición. Los elementos en estas posiciones son 4, 6 y 5 respectivamente.

Padre 2: [2 4 6 8 7 5 3 1]

A continuación buscamos en el primer padre, estos elementos seleccionados y obtenemos que se encuentran en las posiciones cuarta, quinta y sexta.

Padre 1: [1 2 3 4 5 6 7 8]

La descendencia del hijo 1 será igual padre 1 a excepción de las posiciones marcadas:

Hijo 1: [1 2 3 * * * 7 8]

Agregamos los elementos que faltan a la descendencia en el mismo orden en que aparecen en el segundo padre. Es decir, los elementos a añadir son los elementos [4 5 6] pero en el orden en el orden del padre 2 [4 6 5]
Podemos ver el resultado:

Hijo 1: [**1 2 3** 4 6 5 **7 8**]

Para generar el Hijo 2 usaremos el mismo procedimiento partiendo de Padre 1 y con las mismas posiciones aleatorias.

En cuanto a la implementación del cruce:

- Por cada posición hasta el tamaño total de los padres iremos rellenando con true o false una lista de booleanos las posiciones a cambiar mediante un random entre 0 y 1.
- En un bucle por cada posición hasta tamaño total:
    - Si la posición en la lista de booleanos está marcada como true:
        - Buscamos el elemento que hay en esa posición en el Padre 2 y lo añadimos a un vector auxiliar1 para

guardar los elementos marcados en el orden en el que aparecen en el Padre 2.
        - Buscamos el índice en el Padre 1 del elemento obtenido anteriormente del Padre 2.
        - Obtenemos el índice y marcamos a -1 esa posición en el Hijo 1.
        - Buscamos el elemento que hay en esa posición en el Padre 1 y lo añadimos a un vector auxiliar2 para guardar los elementos marcados en el orden en el que aparecen en el Padre 1.
        - Buscamos el índice en el Padre 2 del elemento obtenido anteriormente del Padre 1.
        - Obtenemos el índice y marcamos a -1 esa posición en el Hijo 2.
- En un bucle por cada posición hasta tamaño total:
    - Si la posición en Hijo 1 está marcada a -1:
        - Establecemos, en la posición que nos encontremos, el primer elemento de la lista auxiliar1 de elementos de Padre 2. Posteriormente lo borramos de la lista auxiliar.
    - Si la posición en el Hijo 2 está marcada a -1:
        - Establecemos, en la posición que nos encontremos, el primer elemento de la lista auxiliar2 de elementos de Padre 1. Posteriormente lo borramos de la lista auxiliar.

A continuación podemos ver el pseudocódigo del cruce:

```
Funcion cruceOX2

COMIENZO

    PARA i = 0 HASTA tamTotal HACER
        random = aleatorio.randint(0,1)
        SI random == 0 ENTONCES
            booleanos.add(falso)
        SINO
            booleanos.add(verdadero)
        TERMINASI

    PARA i = 0 HASTA tamTotal HACER
        SI booleanos[i] == verdadero ENTONCES
            hijo1[auxPadre2[i]] = -1
            individuostrue1.add(auxPadre2[i])
            hijo2[auxPadre1[i]] = -1
            individuostrue2.add(auxPadre1[i])
        TERMINASI
    TERMINAPARA




    PARA i = 0 HASTA tamTotal HACER
        SI hijo1[i] == -1 ENTONCES
            hijo1[i] = individuostrue1.eliminarPrimero()
        TERMINASI

        SI hijo2[i] == -1 ENTONCES
            hijo2[i] = individuostrue2.eliminarPrimero()
        TERMINASI
    TERMINAPARA

    escogido1 = hijo1
    escogido2 = hijo2

FIN
```

DESCRIPCIÓN DEL CÓDIGO

ALGORITMO GENÉTICO ESTACIONARIO

ESTRUCTURA DEL ALGORITMO

- Realizamos el torneo binario de selección k=3. Torneo que guardará los vectores ganadores en una Lista Enlazada *elegidosOperadorSelección*.
- Cruzamos los vectores elegidos en el torneo.
- Mutamos los vectores elegidos.
- Incrementamos el número de generaciones cada 2 individuos hijo creados.
- Realizamos el torneo binario de reemplazo k=4. Torneo que guardará los vectores perdedores en una Lista Enlazada *elegidosReemplazo*.
- Mediante un bucle de tamaño del torneo comparamos cada vector de *elegidosOperadorSelección* con el vector de su misma posición en *elegidosReemplazo.*
- Por cada comparación incrementaremos el número de evaluaciones actuales.
    - Por cada vector elegido en el torneo de selección, si el coste del vector elegido n es menor que el coste el vector elegido n para el reemplazo:
        - Metemos el vector elegido en el torneo de selección en la posición que ocupaba el elegido para el reemplazo.
        - Si este individuo mejora al mejor actualizaremos el mejor.

Antes de explicar los torneos, a continuación podemos ver el pseudocódigo del algoritmo principal:

```
Funcion algoritmo, Genetico Estacionario

COMIENZO

    creaPoblacion()
    MIENTRAS numEvaluaciones < totalEvaluaciones HACER
        torneoSeleccion(veces,k)
        cruce()
        torneoReemplazo(vecesReemplazo,kReemplazo)
        PARA i = 0 HASTA vecesReemplazo HACER
            mutaciones(elegidosSeleccion, i)

            numEvaluaciones += 1

            SI coste(elegidosSeleccion[i]) < coste(elegidosReemplazo[i])
                poblacion[posicionesReemplazar[i]] = elegidosSeleccion[i]

                SI coste(elegidosSeleccion < coste(mejor)) ENTONCES
                    mejor = elegidosSeleccion[i]
                TERMINASI
            TERMINASI
        TERMINAPARA
        elegidosSeleccion.limpiar()
        posicionesReemplazar.limpiar()
        elegidosReemplazo.limpiar()
    TERMINAMIENTRAS

FIN
```

TORNEO DE SELECCIÓN

Recibiremos como parámetro la k=3 (número de individuos aleatorios entre los que se elige el mejor) y el número de veces (binario,etc...). Seguidamente, clonaremos la población inicial en una *poblacionAux* para asegurar mejor que no se repitan candidatos, ya que con esta población auxiliar podemos ir borrando a los candidatos elegidos:

- Efectuamos un bucle hasta llegar al número de veces:
    - Este bucle contiene otro bucle hasta llegar al número k de individuos:
        - Lanzamos un número aleatorio entre cero y el tamaño de la población auxiliar y hacemos un get de ese individuo.
        - Si ese individuo tiene un coste menor que el elemento con menor coste actual, lo actualizamos estableciendo el nuevo individuo como el de menor coste.
    - Añadiremos el individuo con menos coste a la lista *elegidosOperadorSelección.*

Pseudocódigo:

```
Funcion torneoSeleccion, AG

COMIENZO

    PARA i = 0 HASTA veces HACER
        menorCoste = +∞
        PARA i = 0 HASTA k HACER
            posRandom = aleatorio.Randint(0, poblacionAux.size()-1)
            individuo = poblacionAux[posRandom]
            SI coste(individuo) < menorCoste ENTONCES
                inviduoMenorCoste = individuo
                menorCoste = coste(inviduoMenorCoste)
            TERMINASI
        TERMINAPARA
        elegidosSeleccion.add(inviduoMenorCoste)
    TERMINAPARA

FIN
```

TORNEO DE REEMPLAZO

Muy parecido al de reemplazo, recibimos como parámetro la k=4 y el número de veces. Seguidamente, clonaremos la población inicial en una *poblacionAux*.

Usaremos los dos mismos bucles que en el anterior torneo pero esta vez iremos guardando el peor de los k individuos, es decir, el de mayor coste.

Iremos añadiendo esos peores individuos a la lista *elegidosReemplazo* y además nos quedaremos con la posición que ocupan dentro de la población en la Lista Enlazada *posicionesReemplazar*.

Pseudocódigo:

```
Funcion torneoReemplazo

COMIENZO

    PARA i = 0 HASTA veces HACER
        mayorCoste = -∞
        PARA j = 0 HASTA k HACER
            posRandom = aleatorio.Randint(0, poblacionAux.size() - 1)
            SI posRandom >= poblacionAux.size() ENTONCES
                posRandom = poblacionAux.size() - 1
            TERMINASI

            individuo = poblacionAux[pos]
            SI coste(individuo) > mayorCoste ENTONCES
                individuoMayorCoste = individuo
                mayorCoste = coste(individuoMayorCoste)
                posMayor = poblacion.indexOf(individuoMayorCoste)
                poblacionAux.remove(posRandom)
            TERMINASI
        TERMINAPARA
        elegidosReemplazo.add(individuoMayorCoste)
        posicionesReemplazar.add(posMayor)
    TERMINAPARA

FIN
```

FUNCIÓN CRUCE

Generamos un número *numElementos* didiviendo (1/ entre la probabilidad de cruce) * 10.
Elegimos un número aleatorio entre 1 y numElementos

- Si ese número aleatorio es menor que *numElemetos* * *probabiliadCruceEstacionario*:
    - Llamamos al operador de cruce con la lista elegidosOperadorSeleccion que encontremos en el fichero de configuración.

Pseudocódigo:

```
Funcion cruce

COMIENZO

    numElementos = Math.round(1 / conf.getProbCruce)*10
    rand = aleatorio.Randint(1, numElementos)

    SI (rand < numElementos * conf.getProbCruce) ENTONCES
        SEGUN conf.getCruce HACER
            CASO OX:
                cruceOX(elegidosSeleccion(0), elegidosSeleccion(1))
                ROMPE
            CASO PMX:
                cruceOX(elegidosSeleccion(0), elegidosSeleccion(1))
                ROMPE
        TERMINASEGUN
    TERMINASI

FIN
```

FUNCIÓN MUTACIONES

Por cada posición del vector:

- Generamos un número *numElementos* didiviendo (1/ constanteMutacion * tamTotal).
- Elegimos un número aleatorio entre 1 y *numElementos*. Si ese número aleatorio es menor que (*numElemetos* constanteMutacion * tamTotal).
    - Añadimos esa posición a una Lista Enlazada llamada *posMuta*, vector que guarda las posiciones elegidas como mutables.
- Si *posMuta* tiene tamaño 2 llamamos al operador mutación con esas dos posiciones y las borramos de la *posMuta.*

Pseudocódigo:

```
Funcion mutaciones

COMIENZO

    PARA i = 0 HASTA i < tamTotal HACER
        numElementos = Math.round(1/conf.getConstanteMutacion * tamTotal)
        rand = aleatorio.Randint(1, numElementos)
        SI rand < numElementos * conf.getConstanteMutacion) ENTONCES
            posMuta.add(i)
        TERMINASI

        SI posMuta.size == 2 ENTONCES
           permutacion(v, posMuta.eliminarPrimero, posMuta.eliminarPrimero)
        TERMINASI
    TERMINAPARA

FIN
```

ALGORITMO GENÉTICO GENERACIONAL

ESTRUCTURA DEL ALGORITMO

En primer lugar generamos la población inicial.

- En primer lugar generamos los n primeros individuos (número de candidatos LRC indicado en el archivo de configuración) con el greedy aleatorizado de la Práctica 1.
- Los demás son generados aleatoriamente hasta llegar a tamaño de población:
    - Por cada posición del vector:
        - Mientras no se haya rellenado esa posición:
            - Generamos un número aleatorio entre 0 y el tamaño total-1.
            - Si no está contenido lo insertamos y marcamos la posición como rellenada.

- Ambos procesos controlan la factibilidad, es decir, que no se repitan los elementos dentro de un mismo vector solución.
- Por cada individuo creado iremos incrementando el número de evaluaciones actuales. Además iremos actualizando una variable que guardará el mejor con el mejor individuo, el de menor coste.

Mientras no se haya alcanzado el número de evaluaciones:

- Se realiza un torneo de selección con k = 2 para obtener aquellos individuos que tienen posibilidad de cruce y pueden pertenecer a la siguiente generación.
- Con una probabilidad de éxito del 70% cruzamos los individuos seleccionados en el paso anterior, de forma que obtenemos dos hijos de cada par de padres.
- Los individuos que no se han cruzado pasan directamente a formar parte de la siguiente generación.
- Posteriormente realizamos la mutación sobre la generación completa.
- Comprobamos que el élite se encuentra sobre la siguiente generación primero a nivel de fitness y posteriormente los genes y en caso de que no

se encuentre en la siguiente generación se sustituye por el peor de los individuos de la generación.
- Finalmente la generación es reemplazada totalmente por la nueva generación.
- Se incrementa el número de evaluaciones, una por cada nuevo individuo calculado.

A continuación podemos ver el pseudocódigo del algoritmo principal:

```
Funcion Algoritmo GG

COMIENZO

    creaPoblacion()
    MIENTRAS numEvaluaciones < numEvaluacionesTotal HACER
        torneoSeleccion(tamañoPoblacion, k)
        cruce()
        PARA i = 0 HASTA tamañoPoblacion HACER
            mutaciones(elegidosSeleccion[i])
        FIN PARA

        SI !contieneElite(elegidosSeleccion) HACER
            pos = buscarPeor(elegidosSeleccion)
            elegidosSeleccion[pos] = elite
        EliteAux = buscarElite(elegidosSeleccion)
        SI coste(EliteAux) < coste(elite) HACER
            elite = EliteAux
        TERMINASI

        poblacion = elegidosSeleccion
        numEvaluaciones += tamañoPoblacion
    TERMINAMIENTRAS

FIN
```

TORNEO DE SELECCIÓN

Recibiremos como parámetro la k=2 (número de individuos aleatorios entre los que se elige el mejor) y el número de veces (En este caso es el número de veces es el tamaño de la población). Seguidamente, clonaremos la población inicial en una *poblacionAux*.
Efectuamos un bucle hasta llegar al número de veces:

- Este bucle contiene otro bucle hasta llegar al número k de individuos:
  - Lanzamos un número aleatorio entre cero y el tamaño de la población auxiliar y hacemos un get de ese individuo.

        - Si ese individuo tiene un coste menor que el elemento con menor coste actual, lo actualizamos estableciendo el nuevo individuo como el de menor coste.
    - Añadiremos el individuo con menos coste a la lista *elegidosOperadorSelección.*

Pseudocódigo:

```
Funcion torneoSeleccion, AG

COMIENZO

    PARA i = 0 HASTA veces HACER
        menorCoste = +∞
        PARA i = 0 HASTA k HACER
            posRandom = aleatorio.Randint(0, poblacionAux.size()-1)
            individuo = poblacionAux[posRandom]
            SI coste(individuo) < menorCoste ENTONCES
                inviduoMenorCoste = individuo
                menorCoste = coste(inviduoMenorCoste)
            TERMINASI
        TERMINAPARA
        elegidosSeleccion.add(inviduoMenorCoste)
    TERMINAPARA

FIN
```

FUNCIÓN CRUCE

Generamos un número *numElementos* didiviendo (1/ entre la probabilidad de cruce) * 10.

Elegimos un número aleatorio entre 1 y *numElementos*

- Si ese número aleatorio es menor que *numElemetos* * *probabiliadCruceEstacionario*:
    - Llamamos al operador de cruce con la lista elegidosOperadorSeleccion que encontremos en el fichero de configuración.

FUNCIÓN MUTACIONES

Por cada posición del vector:

- Generamos un número *numElementos* didiviendo (1/ constanteMutacion * tamTotal).

- Elegimos un número aleatorio entre 1 y *numElementos*. Si ese número aleatorio es menor que (*numElemetos* constanteMutacion * *tamTotal*).
    - Añadimos esa posición a una Lista Enlazada llamada *posMuta*, vector que guarda las posiciones elegidas como mutables.
- Si *posMuta* tiene tamaño 2 llamamos al operador mutación con esas dos posiciones y las borramos de la *posMuta*.

FUNCIONES PARA EL ÉLITE

Contamos con 3 funciones que nos permitan controlar el élite, saber si pertenece a una población y buscar el peor de los individuos de una generación, las funciones son:

- BuscaElite. Busca cual de los individuos es el elite y devuelve su genotipo.
  Pseudocódigo:

```
Funcion contieneElite

COMIENZO

    fitnessElite = coste(elite)
    PARA i = 0 HASTA i < tamañoPoblacion HACER
        SI coste(generacion[i] = fitnessElite) ENTONCES
            DEVUELVE verdadero
        TERMINASI
    TERMINAPARA
    DEVUELVE falso

FIN
```

- ContieneElite. Función que devuelve una variable boleeana indicando si el élite se encuentra o no entre los individuos de una población.
  Pseudocódigo:

```
Funcion contieneElite

COMIENZO

    fitnessElite = coste(elite)
    PARA i = 0 HASTA i < tamañoPoblacion HACER
        SI coste(generacion[i] = fitnessElite) ENTONCES
            DEVUELVE verdadero
        TERMINASI
    TERMINAPARA
    DEVUELVE falso

FIN
```

- BuscaPeor. Devuelve la posición del elemento con mayor coste. El objetivo de esta función es encontrar cual de los individuos de una población tiene mayor coste para poder ser sustituido por el élite en caso de que no se encuentre en esa población.
  Pseudocódigo:

```
Funcion buscaPeor

COMIENZO

   PARA i = 0 HASTA i < tamañoPoblacion hacer
       SI peorCoste < coste(poblacion[i]) ENTONCES
          pos = i
          peorCoste = coste(poblacion[i])
       TERMINASI
    TERMINAPARA
    DEVUELVE pos

FIN
```

ANÁLISIS DEL RESULTADO

Tras guardar en los logs la información del vector solución (coste, generación y vector) cada vez que se mejora iremos forjando las tablas en base a la mejor solución obtenida para cada semilla.

Para cada archivo de Nissan contaremos 5 ejecuciones para las semillas: 26523873, 16720265, 52387326, 72026516 y 87326523. Los parámetros para las ejecuciones son los siguientes:

```
Tamano LRC=5
Candidatos de la GREEDY aleatorizada=5
Tamano poblacion=50
Evaluaciones=50000
Probabilidad Cruce Estacionario=1.0
Probabilidad Cruce Generacional=0.7
Constante para Factor Mutacion Gen=0.001
```

ALGORITMO GENÉTICO ESTACIONARIO OX

Estacionario OX	Tamaño	100	Tamaño	100	Tamaño	150	Tamaño	256	
	Mejor coste	21052466	Mejor coste	1185996137	Mejor coste	498896643	Mejor coste	44759294	
	nissan01		nissan02		nissan03		nissan04		Semillas
	C	Time	*C*	Time	*C*	Time	*C*	Time	
Ejecución 1	22824606,00	282818,00	1321198127,00	279949,00	568504921,00	990491,00	46497392,00	4799604,00	26523873
Ejecución 2	22859306,00	281583,00	1310905163,00	280946,00	566446997,00	990327,00	46257702,00	4899103,00	16720265
Ejecución 3	22911164,00	281808,00	1325499408,00	281825,00	569579847,00	998545,00	46144632,00	4810175,00	52387326
Ejecución 4	22861620,00	282308,00	1336293398,00	279877,00	574002097,00	994482,00	46125778,00	4825857,00	72026516
Ejecución 5	22734416,00	281083,00	1329570995,00	281934,00	568113625,00	992720,00	46310012,00	4837492,00	87326523
Media	8,48 %	281920,00	11,69 %	280906,20	14,12 %	993313,00	3,37 %	4834446,20	
Desv. típica	0,31 %	667,76	0,80 %	984,59	0,57 %	3390,13	0,34 %	38944,26	

Conforme vamos aumentando el tamaño de los vectores el tiempo de ejecución va creciendo de forma exponencial.

Además, este algoritmo es más robusto en los archivos *nissan01* y *nissan04*, ya que como podemos observar, encontramos las menores desviaciones en estos casos. Las soluciones más cercanas son nuevamente las *nissan01* y *nissan04*, ambas que tienen la media más cercana a 0, sobretodo *nissan04*.

ALGORITMO GENÉTICO ESTACIONARIO PMX

Estacionario PMX	Tamaño	100	Tamaño	100	Tamaño	150	Tamaño	256	
	Mejor coste	21052466	Mejor coste	1185996137	Mejor coste	498896643	Mejor coste	44759294	
	nissan01		nissan02		nissan03		nissan04		Semillas
	C	Time	*C*	Time	*C*	Time	*C*	Time	
Ejecución 1	22584856,00	278784,00	1270244554,00	282308,00	559947200,00	988959,00	47164392,00	4661766,00	26523873
Ejecución 2	22496178,00	280489,00	1273529506,00	282437,00	551515508,00	995776,00	47492740,00	4664293,00	16720265
Ejecución 3	22605252,00	279626,00	1301522363,00	280740,00	561068593,00	1005274,00	46688436,00	4705440,00	52387326
Ejecución 4	22589456,00	278672,00	1275617851,00	282139,00	563667569,00	1006345,00	46933636,00	4757288,00	72026516
Ejecución 5	22642732,00	281663,00	1283708164,00	281166,00	551420396,00	999837,00	46399776,00	4719837,00	87326523
Media	7,27 %	279846,80	8,00 %	281758,00	11,75 %	999238,20	4,86 %	4701724,80	
Desv. típica	0,26 %	1251,94	1,06 %	757,54	1,14 %	7157,95	0,94 %	40084,19	

Al igual que ocurre en el caso anterior al aumentar el tamaño de los vectores el tiempo de ejecución aumenta exponencialmente, nuevamente en los archivos *nissan01* y *nissan04* son los que menor desviación tienen.

ALGORITMO GENÉTICO GENERACIONAL OX2

Generacional OX2	Tamaño	100	Tamaño	100	Tamaño	150	Tamaño	256	
	Mejor coste	21052466	Mejor coste	1185996137	Mejor coste	498896643	Mejor coste	44759294	
	nissan01		nissan02		nissan03		nissan04		Semillas
	C	Time	*C*	Time	*C*	Time	*C*	Time	
Ejecución 1	23034752,00	159456,00	1365778580,00	161244,00	584547323,00	563065,00	47074924,00	2333334,00	26523873
Ejecución 2	23050142,00	161545,00	1378108830,00	161079,00	579543520,00	558764,00	46538912,00	2394224,00	16720265
Ejecución 3	23088906,00	161345,00	1344826241,00	162688,00	586026169,00	557661,00	46960522,00	2392328,00	52387326
Ejecución 4	23036814,00	160472,00	1367109460,00	160635,00	586814607,00	560316,00	47044314,00	2460498,00	72026516
Ejecución 5	22982938,00	160546,00	1382134540,00	160621,00	581154647,00	557481,00	47001098,00	2421974,00	87326523
Media	9,43 %	160672,80	15,31 %	161253,40	16,98 %	559457,40	4,84 %	2400471,60	
Desv. típica	0,18 %	829,09	1,22 %	847,19	0,63 %	2311,17	0,49 %	46578,78	

Nuevamente observamos que cada vez que aumenta el tamaño de los vectores, su tiempo de ejecución crece de forma exponencial.

Este algoritmo es más robusto en los archivos *nissan01* y *nissan04*, siendo de nuevo, el archivo nissan04 el que más se aproxima a la solución global.

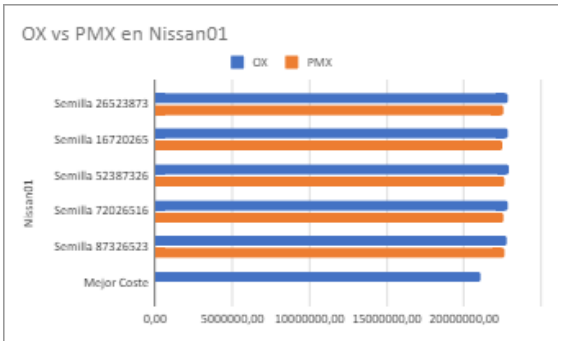
ALGORITMO GENÉTICO GENERACIONAL PMX

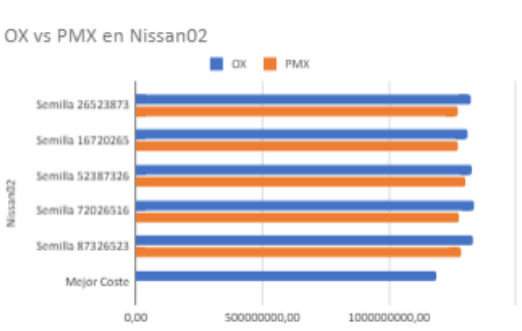
Generacional PMX	Tamaño	100	Tamaño	100	Tamaño	150	Tamaño	256	
	Mejor coste	21052466	Mejor coste	1185996137	Mejor coste	498896643	Mejor coste	44759294	
	nissan01		nissan02		nissan03		nissan04		Semillas
	C	Time	*C*	Time	*C*	Time	*C*	Time	
Ejecución 1	22943268,00	167708,00	1353469286,00	163725,00	588597216,00	543968,00	47070696,00	2487480,00	26523873
Ejecución 2	22898104,00	161089,00	1358446978,00	165025,00	581791147,00	565499,00	47175192,00	2487969,00	16720265
Ejecución 3	22910484,00	164286,00	1371303148,00	164248,00	584651209,00	557467,00	46770434,00	2456173,00	52387326
Ejecución 4	23065570,00	162309,00	1370328547,00	160861,00	578024907,00	556363,00	46437808,00	2456807,00	72026516
Ejecución 5	23024292,00	163835,00	1321044008,00	158063,00	589568645,00	568821,00	46383348,00	2439358,00	87326523
Media	9,10 %	163845,40	14,24 %	162384,40	17,16 %	558423,60	4,49 %	2465557,40	
Desv. típica	0,35 %	2503,61	1,72 %	2882,63	0,96 %	9647,44	0,80 %	21412,18	

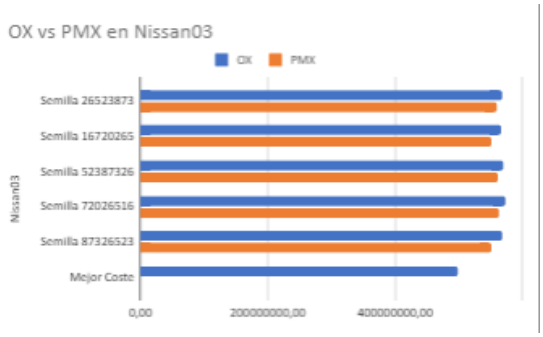
En esta tabla observamos que ocurre lo mismo que lo descrito en las anteriores.

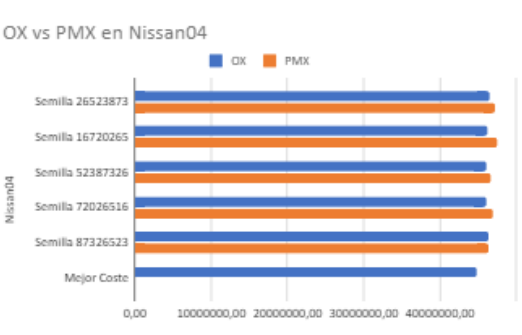
COMPARATIVA

ALGORITMO GENÉTICO ESTACIONARIO OX VS PMX

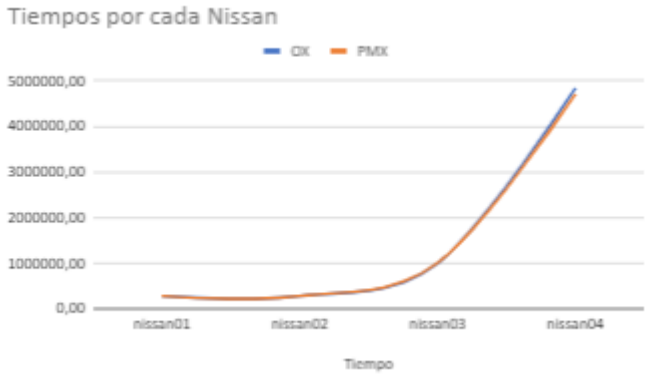




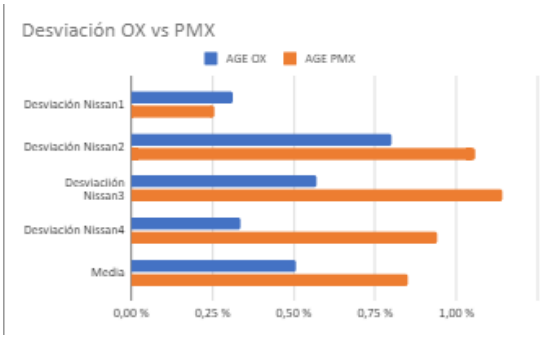




Tras realizar las ejecuciones del cruce PMX y OX para las diferentes semillas en los diferentes archivos, podemos observar que el cruce PMX es algo mejor en comparación al cruce OX. A excepción del fichero *nissan04* donde es ligeramente mejor el cruce OX. Por otro lado, su tiempo de ejecución es bastante similar.



El tiempo de ejecución se eleva de forma exponencial dependiendo del tamaño del fichero para ambos cruces, ligeramente como se observa en la gráfica el tiempo de ejecución del cruce PMX es algo más rápido que el OX, pero realmente resulta casi insignificante.

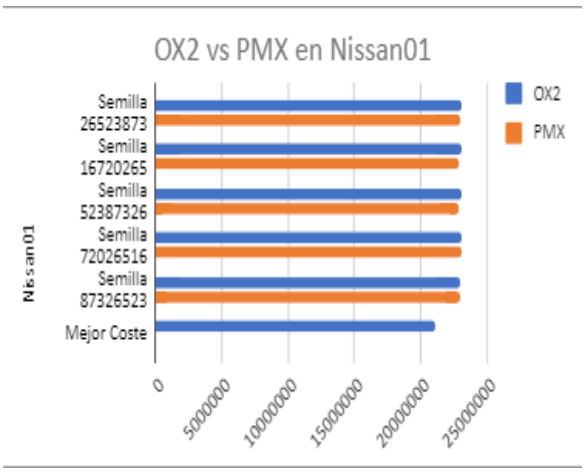


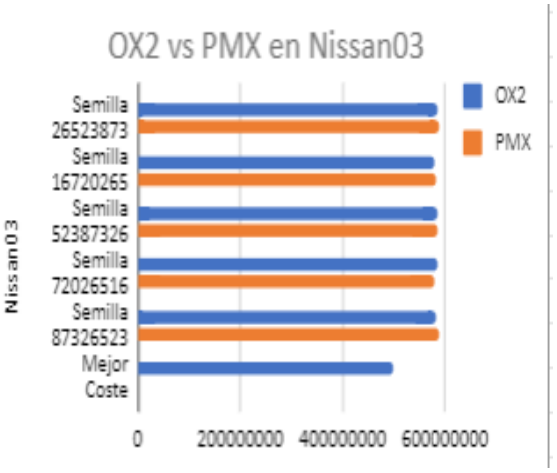
Para ambos cruces, se trata de un algoritmo bastante robusto, ya que su desviación típica es bastante baja para todos los casos. Aunque el OX es más robusto, obtenemos soluciones más cercanas al mejor con el cruce PMX.

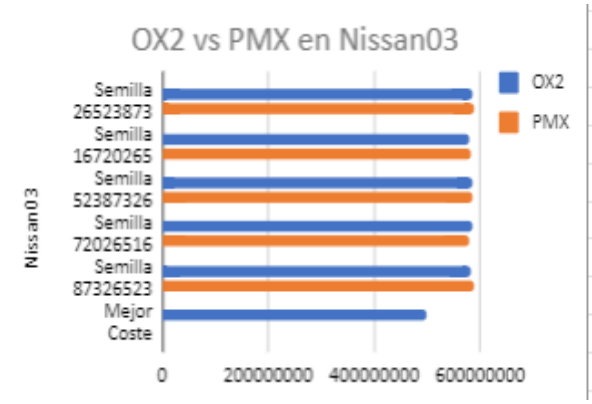
	nissan01		nissan02		nissan03		nissan04		MEDIA	
	C	Time	*C*	Time	*C*	Time	*C*	Time	*C*	Time
AGE OX	8,48 %	281920,00	11,69 %	280906,20	14,12 %	993313,00	3,37 %	4834446,20	9,42 %	1597646,35
AGE PMX	7,27 %	279846,80	8,00 %	281758,00	11,75 %	999238,20	4,86 %	4701724,80	7,97 %	1565641,95

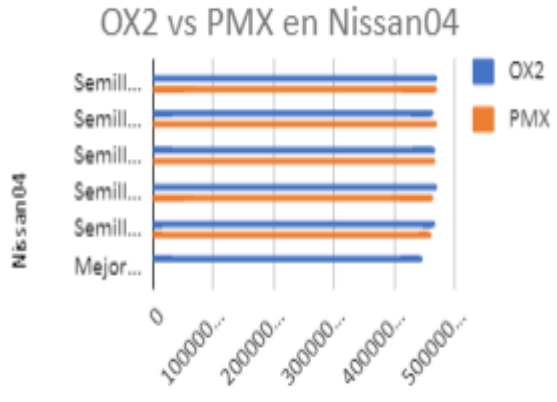
Al encontrarnos con un tiempo de ejecución parecido, nos quedamos con el que encuentra soluciones más cercanas, el cruce PMX.

ALGORITMO GENÉTICO GENERACIONAL OX2 VS PMX









Una vez realizadas todas las ejecuciones vemos que ambos cruces están realmente muy empatados , ya que sus resultados son muy similares, en este caso el más cercano a obtener la mejor de las soluciones vuelve a ser el fichero *nissan04*.

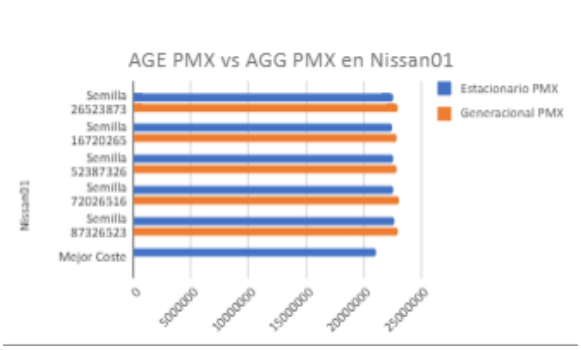
En cuanto a la desviación, en el cruce PMX vuelve a ser ligeramente mayor en comparación al cruce OX2, pero aún así podemos observar que se trata de un algoritmo muy robusto, ya que su desviación no supera el 2%.

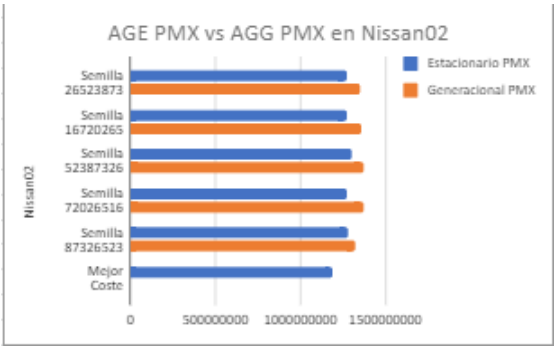
Los tiempos de ejecución vuelven a ser muy similares, ya que tardan prácticamente lo mismo en realizar la ejecuciones con cada semilla, nuevamente se observa un crecimiento exponencial.

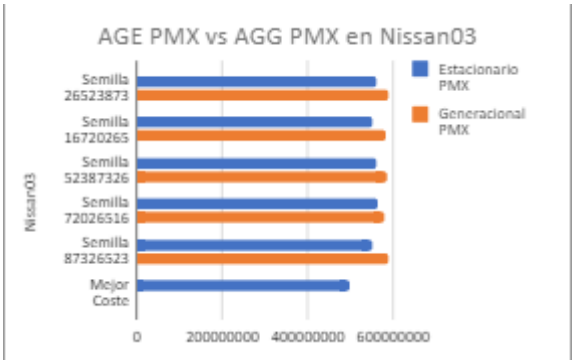
	nissan01		nissan02		nissan03		nissan04		MEDIA	
	C	Time	*C*	Time	*C*	Time	*C*	Time	*C*	Time
AGE OX2	9,43 %	160672,80	15,31 %	161253,40	16,98 %	559457,40	4,84 %	2400471,60	11,64 %	820463,80
AGE PMX	9,10 %	163845,40	14,24 %	162384,40	17,16 %	558423,60	4,49 %	2465557,40	11,25 %	837552,70

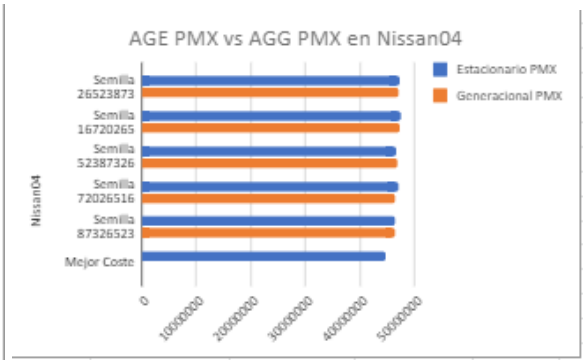
Comparando la tabla de resultados observamos que ambos cruces son igual de eficaces, ya que su tiempo de ejecución es bastante similar y su proximidad a la solución igual, por lo que podríamos decir que los dos son igual de buenos y ambos nos servirían ya que sus resultados son muy similares. Sin embargo el PMX encuentra ligeramente soluciones más cercanas.

MEJOR ESTACIONARIO VS MEJOR GENERACIONAL



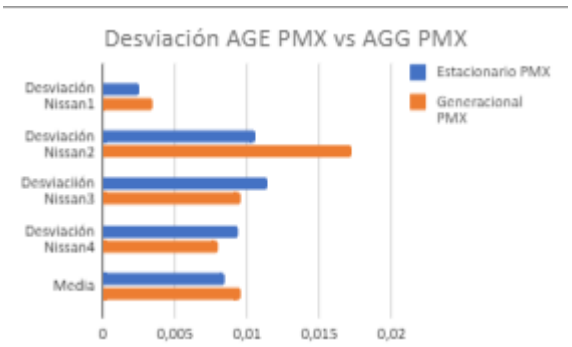






Una vez definido cual es el mejor de cruces, el PMX, comparamos ambos algoritmos y observamos que en todos los ficheros es mejor el algoritmo estacionario, a excepción del fichero *nissan04*, donde se encuentran prácticamente empatados ambos algoritmos.

Si observamos su gráfica de tiempo empleado vemos que el algoritmo generacional con cruce PMX es mucho menor, esto se debe a que en el algoritmo estacionario se producen más generaciones, por tanto su tiempo de cómputo aumenta considerablemente.



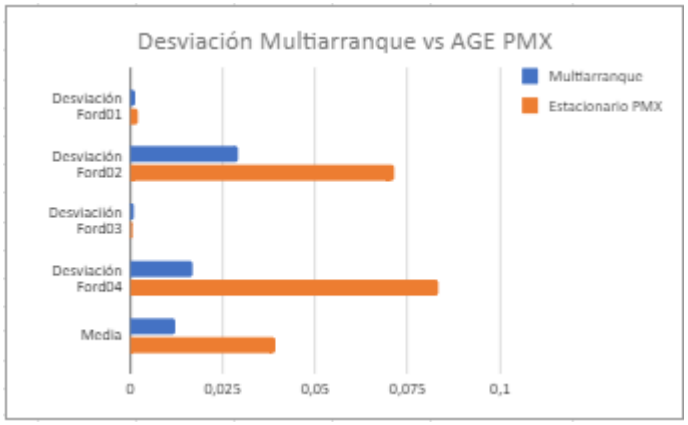
Nuevamente comparando la gráfica con las distintas desviaciones vemos que nuestros algoritmos son muy robustos, ya que su desviación es bastante baja, como hemos mencionado anteriormente. Observamos que el fichero con el que es más robusto es el *nissan01* y el archivo con el que los algoritmos tienen menos robustez es el *nissan02*.

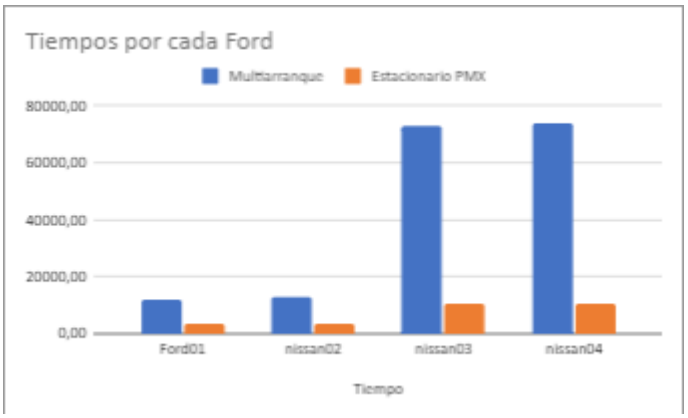
	nissan01		nissan02		nissan03		nissan04		MEDIA	
	C	Time	*C*	Time	*C*	Time	*C*	Time	*C*	Time
Estacionario PMX	7,27 %	279846,80	8,00 %	281758,00	11,75 %	999238,20	4,86 %	4701724,80	7,97 %	1565641,95
Generacional PMX	9,10 %	163845,40	14,24 %	162384,40	17,16 %	558423,60	4,49 %	2465557,40	11,25 %	837552,70

Finalmente en la tabla que se encuentran recogidos los datos más importantes, podemos llegar a la conclusión de que el algoritmo genético estacionario es el que nos produce las soluciones más cercanas al óptimo global a cambio de prácticamente el doble de tiempo de cómputo, en estos casos se debería de valorar si necesitamos un algoritmo que nos aporte unas soluciones decentes por un tiempo de ejecución decente o las mejores posibles a cambio de un tiempo de cómputo mucho más alto.

MEJOR PRÁCTICA 1 VS MEJOR PRÁCTICA 2

Finalmente al comparar las ejecuciones en los archivos Ford con el ganador de la práctica 1 y el ganador de la práctica actual, observamos en los resultados que son bastante similares, pero en algunos casos el algoritmo multiarranque obtiene mejores resultados que el algoritmo estacionario.





En cuanto a los tiempos de ejecución, el algoritmo multiarranque es el que los tiene más elevados, y en cuanto a la desviación ambos algoritmos demuestran nuevamente su robustez, ya que no llega al 1% de desviación.

	ford01		ford02		ford03		ford04		MEDIA	
	C	Time	*C*	Time	*C*	Time	*C*	Time	*C*	Time
Multiarranque	2,26 %	12143,80	3,12 %	12763,00	1,72 %	72910,80	1,20 %	73806,80	2,07 %	42906,10
Estacionario PMX	2,82 %	3623,80	12,60 %	3558,40	1,94 %	10828,00	3,73 %	10713,20	5,27 %	7180,85

Como conclusión vemos que el algoritmo multiarranque es mucho mejor en la obtención de soluciones globales, ya que es un algoritmo de propósito específico, en cambio los algoritmos genéticos son buenos para problemas de propósito general, donde no necesitemos forzosamente la mejor de las soluciones, se observa en la última tabla como se cumple lo anteriormente mencionado, ya que el algoritmo multiarranque tiene una media de 2.07% bastante cercana a los óptimos locales, y por otro lado el algoritmo estacionario tiene una media de 5.27%, lo que nos aporta también soluciones bastante buenas.