Grado en Ingeniería Informática

Metaheurística

Curso 2021/2022

Universidad de Jaén

Informe práctica 2

- Estacionario PMX / OX
-Generacional PMX / OX2

Adrián Arboledas Fernández
Grupo: 2
aaf00022@red.ujaen.es
26523518X

José Antonio Morales Velasco
Grupo: 2
jamv0007@red.ujaen.es
76593264K

Descarga Logs

UNIVERSIDAD DE JAÉN

ÍNDICE

INTRODUCCIÓN

Descripción del problema.

El problema consiste en “minimizar el coste” de distribución de piezas entre fábricas, asignando a los mayores flujos de piezas, las menores distancias entre departamentos. Para ello dado un conjunto ***p*** de ***P*** departamentos hemos de seleccionar un conjunto ***l*** de ***L*** localizaciones, de manera que la distribución de piezas entre departamentos sea lo más eficiente posible.

Definición del problema.

- $Sea\ p = \left(p_i,\ i=1,\ \ldots,\ n\right)$, departamentos.
- Sea $\mathrm{l} = \left(\mathrm{l_j},\ \mathrm{j}=1,\ \ldots,\ \mathrm{n}\right)$, localizaciones.
- Se definen dos matrices $\mathrm{F} = \left(\mathrm{f_{ij}}\right)$ y $\mathrm{D} = \left(\mathrm{d_{ij}}\right)$, de dimsensión cuadrada **n**.
- F: matriz de flujo de piezas, siendo $\mathrm{f_{ij}}$, el número de piezas que pasan del departamento i al j.
- D: matriz de distancia, siendo $\mathrm{d_{ij}}$, la distancia entre los departamentos i, j.
- El coste de asignar $\mathrm{p_i}$ a $\mathrm{l_k}$ y $\mathrm{p_j}$ a $\mathrm{l_l}$ es : $\left(\mathrm{f_{ij}}\cdot\ \mathrm{d_{kl}} + \mathrm{f_{ji}}\cdot\ \mathrm{d_{lk}}\right)$.

Función Objetivo.

La función objetivo del problema consiste en minimizar el coste de asignación de los departamentos con las distintas localizaciones.

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

$$\sum_{j=1}^{n} x_{ij} = 1 \quad 1 \leq i \leq n,$$

$$x_{ij} \in \{0,1\} \quad 1 \leq i \leq n$$

Representación de la solución

Como representación de la solución se utilizará un vector de permutaciones del conjunto ***N,*** donde el índice de dicho vector corresponderá al flujo de piezas y el valor contenido será el departamento.

Por tanto para la implementación del vector de permutaciones, se usará un vector de enteros de tamaño ***N***, tal que: int [ ] vectorSolucion.

ALGORITMOS PARA LA RESOLUCIÓN DEL PROBLEMA

En este apartado se explicarán cómo se han implementado los distintos algoritmos para la resolución del problema:

PARÁMETROS DEL PROBLEMA:

Para la ejecución de los algoritmos que veremos a continuación hemos utilizado la siguiente lista de parámetros:

1. **Archivos:** Los diferentes archivos con los que trabajará cada algoritmo.
2. **Semillas:** Contendrá las 5 semillas utilizadas para la aleatoriedad de los algoritmos.
3. **Algoritmos:** Contiene los cuatro algoritmos a ejecutar, Generacional OX2/PMX y Estacionario OX/PMX.
4. **TamPoblacion:** Se le asigna el tamaño de la población.
5. **MaxEvaluaciones:** Número máximo de evaluaciones de la población.
6. **probEstacionario:** Probabilidad de cruce del Algoritmo estacionario.
7. **probGeneracional:** Probabilidad de cruce del Algoritmo Generacional.
8. **probMutacion:** Probabilidad de mutación de cada gen de un individuo.
9. **tamTorneoSeleccionAGE:** *"K"* correspondiente al torneo de selección del algoritmo estacionario.

10. **tamTorneoRemplazamientoAGE:** "K" correspondiente al torneo de reemplazamiento del algoritmo estacionario.

11. **tamTorneoSeleccionAGG:** *"K"* correspondiente al torneo de selección del algoritmo generacional.

12. **TamLRC:** Tamaño de la lista restringida de candidatos.

13. **numIndividuosEstacionario:** Cantidad de individuos a seleccionar en estacionario.

14. **rondasTorneoEstacionario:** Cantidad rondas torneo estacionario.

CREACIÓN POBLACIÓN INICIAL

La población inicial de la que van a partir los distintos algoritmos se genera en dos fases:

1. **Creación Lista restringida de candidatos(LRC):**
   Se calculan mediante un algoritmo **Greedy aleatorizado**, del cual se seleccionan los cinco mayores flujos y las cinco menores distancias.
   A partir de estos se construyen cinco individuos de forma aleatoria, evitando que se repitan índices y realizando un intercambio 2-opt.

   a. **Greedy aleatorizado:** Basándonos en la solución obtenida en nuestro algoritmo Greedy generamos cinco soluciones que serán construidas de la siguiente manera:

      i. Cogemos las 5 unidades con mayor flujo, y seleccionamos una de ellas mediante un aleatorio.

      ii. Cogemos las 5 distancias más concéntricas, y seleccionamos una de ellas mediante un aleatorio.

      iii. Se asigna la localización del paso 2 a la unidad escogida del paso 1

      El procedimiento se repite de forma constructiva hasta generar todas las soluciones de la población LRC completa, evitando repeticiones.

b. **Generación aleatoria:** Se calculan los 45 individuos restantes de forma totalmente aleatoria sin repetir permutaciones dentro del individuo.

ESTRUCTURA BASE DEL ALGORITMO

La siguiente estructura es usada para la implementación de cualquier algoritmo genético, en nuestro caso para el Estacionario y el generacional.

```
AlgoritmoGenetico()
        Begin
                t = 0
                inicializar P(t)
                evaluar P(t)
                WHILE (no cumpla condición de parada ) do
                        t = t+1
                        seleccionar p' desde p(t-1)
                        Cruce p'
                        mutar P'
                        evaluar P(t)
                        reemplazar p(t) a partir de p(t-1) y p`
                endWHILE
        END
```

Al principio del algoritmo se realiza una evaluación de la población completa, independientemente del modelo usado.

MODELO ESTACIONARIO

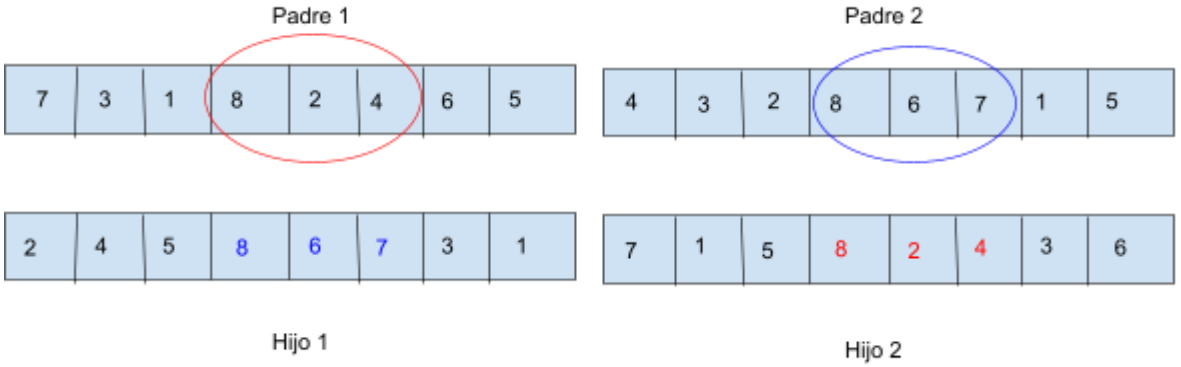
El algoritmo Estacionario es un tipo de algoritmo genético, donde en cada iteración se seleccionan algunos padres(los que más se adaptan) y se le aplican los operadores genéticos.

Podemos dividir el modelo estacionario en 4 partes

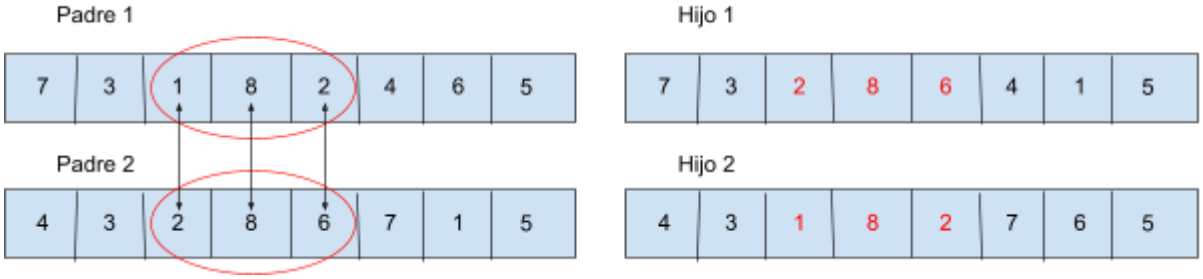
1. **Selección:** Para la selección de individuos realizamos un torneo con un ***"k = 3"*** con dos rondas, para cada ronda se escogen aleatoriamente 3 individuos distintos, y de estos tres nos quedamos con el que mejor se adapta (menor fitness), obteniendo al final dos individuos, uno en cada ronda.

2. **Cruce:** Se aplica el operador de cruce,donde se combinan los individuos seleccionados para generar nuevos hijos, en nuestro caso aplicamos dos:

a. **OX:** En este operador de cruce, se escogen dos genes aleatorios, y el intervalo resultante entre ambos genes, con respecto al padre 1 se rellena en el hijo 2 y el del padre 2 en el hijo 1.
El resto del individuo se rellena recorriendo el padre 1 en el caso del hijo 1 o el padre 2 en caso del hijo 2 y para cada valor, si no está añadido en el fragmento intercambiado, se añade en esa posición; sino se salta al siguiente elemento del padre, así hasta rellenar los individuos hijos. Se empieza a rellenar a partir del siguiente gen del fragmento intercambiado. Es decir en este ejemplo, se empieza rellenar en el elemento con la posición 6 (el primero elemento es el 0) y se va comprobando desde el primer elemento, 7. Como el 7 está, se pasa al siguiente, que en este caso es el 3, y que actualmente no está por lo que se añade.



b. **PMX:** Funcionamiento similar al cruce OX, escogemos dos genes aleatorios del individuo, obteniendo así un intervalo definido por ambos, el intervalo del padre 1, se copia en las posiciones correspondientes del hijo 2 y el intervalo del padre 2 se copia en las posiciones correspondientes del hijo 1.
La diferencia respecto al *¨OX¨* es que almacenamos las correspondencias, es decir si en el padre 1 obtenemos el intervalo (1,8,2) y en el padre 2 (2,8,6) se relacionarían de la siguiente manera 1←→2, 8←→8, 2←→ 6, por tanto al rellenar los hijos si el valor a añadir coincide con alguno perteneciente al intervalo, se busca su correspondencia, abajo se muestra un ejemplo explicativo.



3. **Mutación:** Una vez tenemos a los individuos cruzados, es hora de realizar la mutación, para cada gen del individuo, se lanza un valor aleatorio, el cual dota de probabilidad de mutación a cada gen.
   Si el aleatorio se encuentra dentro de la probabilidad de mutación, realizamos un intercambio aleatorio **2-opt**.

4. **Reemplazamiento:** Como última etapa del algoritmo, necesitamos reemplazar los dos individuos, que han sido seleccionados, cruzados y mutados, de manera que evolucione la población siempre que los individuos se adapten mejor que los obtenidos al realizar el torneo.

   Para obtener los individuos a reemplazar, se realiza para cada uno de los individuos anteriores un torneo con ***"k=4"***, donde se obtienen 4 individuos de forma aleatoria de la población actual.

   De estos cuatro individuos, nos quedamos con el que peor se adapta, el cual va a ser comparado con uno de los individuos anteriores. Solo se reemplaza el individuo de la población actual por el obtenido durante el proceso de cruce y mutación, si este se adapta mejor.

MODELO GENERACIONAL

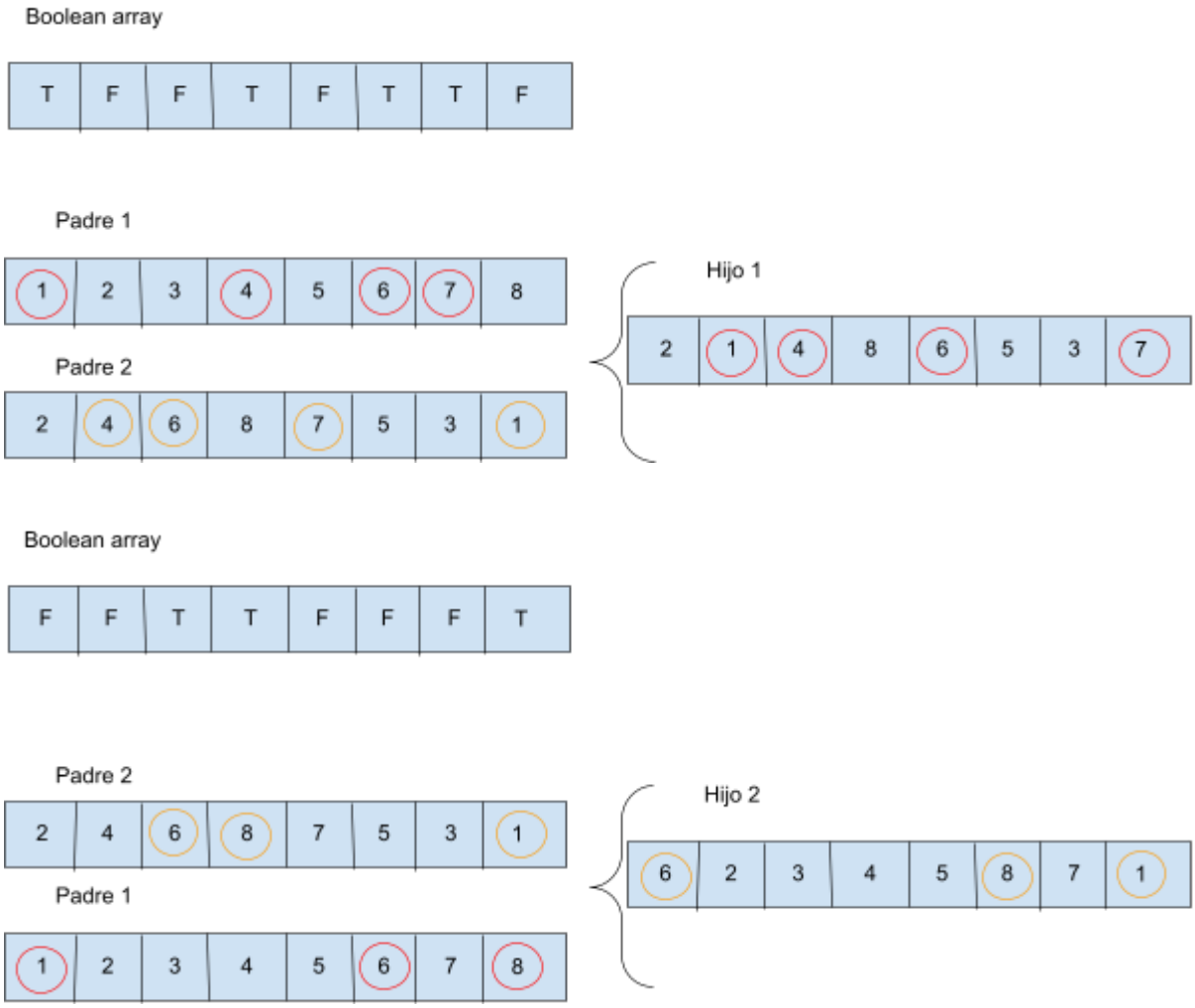
El modelo generacional, es un tipo de algoritmo genético en el cual, a partir de la población actual, se generan un número de hijos de tamaño equivalente a la población actual, se aplican los operadores, mutación y reemplazan la población anterior. Este modelo cuenta con elitismo, es decir para cada generación se sabe cual es el mejor individuo de la población, de tal forma que si este no está en la nueva población se añade a esta.

El modelo generacional también consta de 4 fases:

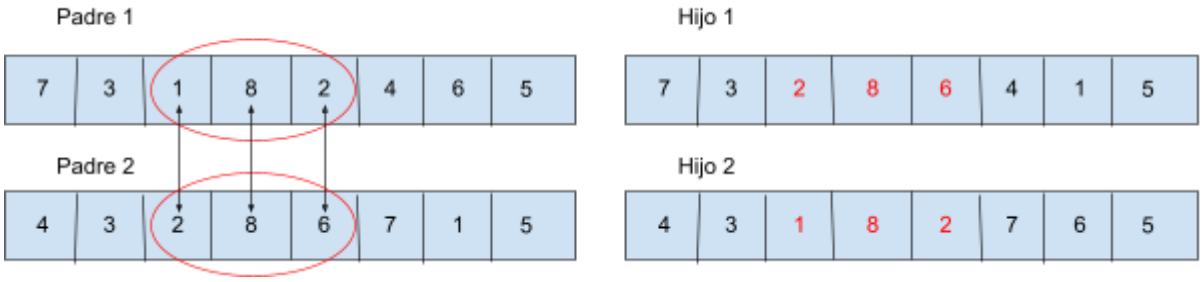
1. **Selección:** La selección se realiza con un torneo binario con **"k=2"** de forma que para toda la población, se escogen dos individuos de forma aleatoria y nos quedamos con el mejor de ambos. Esto se repite hasta generar un número de candidatos igual al tamaño de la población actual.

2. **Cruce:** Para los individuos seleccionados durante el torneo de selección, se aplica un cruce entre cada par de individuos, siempre que se cumpla la probabilidad de cruce que hay impuesta. Los operadores de cruce que se han implementado son los siguientes:

a. **OX2:** En el operador OX2, se realiza un cruce bastante parecido al operador OX, de forma que, primero para cada gen del individuo del padre 1 se lanza un aleatorio booleano de forma que se obtiene un vector que indica con true los genes a recibir para el hijo 1 de este padre. Para el padre 2 se buscan los valores del gen que corresponde con los seleccionados en el padre 1. Una vez hecho esto, el hijo 1 se rellena con los genes del padre 2 no seleccionados. Posteriormente los genes restantes se rellenan con los valores seleccionados del padre 1 en orden, es decir, si hay un hueco, se rellena con el primer elemento a true, el segundo con el segundo, y así sucesivamente. Una vez hecho esto con el padre 1 se repite con el 2 para generar otro individuo.



b. **PMX:** Funcionamiento similar al cruce OX, escogemos dos genes aleatorios del individuo, obteniendo así un intervalo definido por ambos, el intervalo del padre 1, se copia en las posiciones correspondientes del hijo 2 y el intervalo del padre 2 se copia en las posiciones correspondientes del hijo 1.
La diferencia respecto al *¨OX¨* es que almacenamos las correspondencias, es decir si en el padre 1 obtenemos el intervalo (1,8,2) y en el padre 2 (2,8,6) se relacionarían de la siguiente manera 1←→2, 8←→8, 2←→ 6, por tanto al rellenar los hijos si el valor a añadir coincide con alguno perteneciente al intervalo, se busca su correspondencia, abajo se muestra un ejemplo explicativo.



3. **Mutación:** Una vez tenemos a los individuos cruzados, es hora de realizar la mutación, para cada gen del individuo, se lanza un valor aleatorio, el cual dota de probabilidad de mutación a cada gen.
Si el aleatorio se encuentra dentro de l a probabilidad de mutación, realizamos un intercambio aleatorio **2-opt**.

4. **Reemplazamiento:** A diferencia del estacionario, aquí no se realiza ningún torneo, puesto que aquí reemplazamos la población completa.
Aun así hay que tener en cuenta el factor del elitismo y es que en la población generada, debemos de almacenar siempre el élite de la población anterior.
Para ello, lo primero que hacemos es buscar por *"fitness"*, para comprobar si se encuentra en la población.

   Si no se encuentra se elimina el individuo que peor se adapte de la población y añadimos el elite, pero en el caso de encontrarlo, para asegurarnos a 100% que ese individuo es nuestro élite, debemos de comprobar gen a gen, en el caso de que sea el mismo, no se añadiría y se reemplazaría la población completa, si no es el mismo se realiza lo descrito anteriormente.

PSEUDOCÓDIGO

El pseudocódigo para los diferentes operadores de cruce son los siguientes:

```
OperadorCruceOX()
        Begin
                a1,a2 = generacionIntervaloAleatorio;
                int[ ] hijo1,hijo2;
                FOR  i = a1 to a2 do
                        hijo1[i] = padre1[i];
                        hijo2[i] = padre2[i];
                endFor

                boolean llenoH1,llenoH2;
                int contadorPosicionH1,contadorPosicionH2 = a2;
                int contEleHijo1,contEleHijo2 = a2-a1;

                FOR i = 0 until !llenoH1 && !llenoH2 do

                        boolean estaH1, estaH2

                        FOR j = 0 to a2 do
                                if padre2[ j ] == padre1 [ j ] then
                                        estaH1 = true
                                endif
                        endFor

                        FOR j = 0 to a2 do
                                if padre1[ j ] == padre2 [ j ] then
                                        estaH2 = true
                                endif
                        endFor
```

```
            if !estaH1 then
                if !llenoH1 then
                    hijo1[ contadorPosicionHijo1 ] = padre2 [ i ]
                    contadorPosicionHijo1++
                    contEleHijo1++;

                    if contEleHijo1 == tamañoHijo1 then
                        llenoH1 = true
                    endif
                endif
            endif

            if !estaH2 then
                if !llenoH2 then
                    hijo2[ contadorPosicionHijo2 ] = padre1 [ i ]
                    contadorPosicionHijo2++
                    contEleHijo2++;

                    if contEleHijo2 == tamañoHijo2 then
                        llenoH2 = true
                    endif
                endif
            endif


        endFor

        añadirHijos(hijo1,hijo2);

Return Hijos;

END
```

```
OperadorCrucePMX()
        Begin
                For i = 0 to n  do
                        a1,a2 = generacionIntervaloAleatorio;

                        For j = a1 to a2 do
                                hijo1[j] = padre(i+1)[j]
                                hijo2[j] = padre(i)[j]
                                correspondencia1.add(padre(i)[j], padre(i+1)[j]);
                                correspondencia2.add(padre(i+1)[j], padre(i)[j]);
                        endFor

                        For j = 0 to n do
                                //hijo1
                                valor1 = padre(i)[j]
                                cont = 0
                                While(encontrado) do
                                        if(valo1r == correspondencia1[cont].key) then
                                                valor1 = correspondencia1[cont].value

                                        else
                                                cont++
                                        endIF

                                        if (cont == tamCorrespondencia1) then
                                                encontrado = false
                                        endIF
                                endWhile
                                hijo1[j] = valor1;

                                valor2 = padre(i+1)[j]
                                cont2 = 0
```

```
//hijo2
                    valor2 = padre(i+1)[j]
                    cont2 = 0
                    While(encontrado) do
                        if(valor2 == correspondencia2[cont2].key) then
                            valor2 = correspondencia2[cont2].value

                        else
                            cont2++
                        endIF

                        if (cont2 == tamCorrespondencia2) then
                            encontrado = false
                        endIF
                    endWhile
Return hijos
END
```

```
OperadorCruceOX2()
        Begin
                hijos[ ]
                boolean posicionesPadre1[ padre1.length ]
                boolean posicionesPadre2[ padre2.length ]
                int valores1[ ]
                int valores2[ ]

                FOR i = 0 to posicionesPadre1.length do
                        posicionesPadre1[ i ] = aleatorio.nextBool()
                        if posicionesPadre1[ i ] then
                                valores1.add( padre1[ i ])
                        endif
                        posicionesPadre2[ i ] = aleatorio.nextBool()
                        if posicionesPadre2[ i ] then
                                valores2.add( padre2[ i ])
                        endif
                endFor

                int[ ] hijo1 = -1
                int[ ] hijo2 = -1

                FOR i = 0 to padre2.length do
                        bool coincide = false
                        FOR j = 0 to valores1.size && !coincide do
                                if padre2[ i ] == valores[ k ] then
                                        coincide = true
                                endif
                        endFor

                        if !coincide then
                                hijo1 [ j ] = padre2 [ j ]
                        endif
                endFor
```

```
        FOR i = 0 to padre1.length do
                bool coincide = false
                FOR j = 0 to valores2.size && !coincide do
                        if padre1[ i ] == valores2[ k ] then
                                coincide = true
                        endif

                endFor

                if !coincide then
                        hijo2 [ j ] = padre1 [ j ]
                endif

        endFor

        FOR i = 0 to hijo1.length do
                if hijo1[ i ] == -1 then
                        hijo1 [ i ] = valores1.remove(0)
                endif

                if hijo2[ i ] == -1 then
                        hijo2 [ i ] = valores2.remove(0)
                endif

        endFor

        hijos = addHijos(hijo1,hijo2)

Return hijos

End
```

CONCLUSIONES

ANÁLISIS COMPARATIVO

Promedio coste y tiempo

	Nissan01		Nissan02		Nissan03		Nissan04		MEDIA	
	C	Time	*C*	Time	*C*	Time	*C*	Time	*C*	Time
Estacionario OX	12,01 %	2117,80	**32,40 %**	**1828,20**	24,93 %	**3501,20**	11,03 %	**4408,99**	20,09 %	**2964,05**
Estacionario PMX	**11,97 %**	**2023,80**	**32,40 %**	2104,80	24,75 %	3896,60	11,22 %	10848,00	20,08 %	4718,30
Generacional OX2	12,09 %	2155,00	**32,40 %**	2158,40	**24,70 %**	4300,00	**10,86 %**	11089,80	**20,01 %**	4925,80
Generacional PMX	12,08 %	2079,00	**32,40 %**	2104,80	24,80 %	3896,60	11,30 %	10611,00	20,14 %	4672,85

En esta tabla se recogen el promedio de costes para cada uno de los archivos y el tiempo de ejecución medio de cada archivo.
En la columna de medias se muestra la media de la desviación típica de coste y la media del tiempo de cada algoritmo respecto a la ejecución de los cuatro archivos.

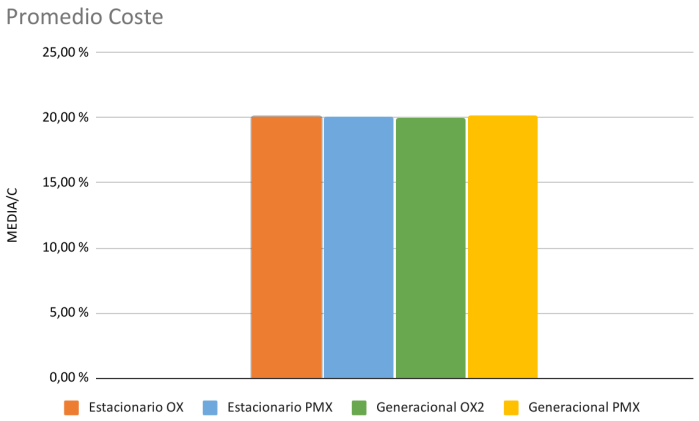
- **Nissan01:** Para el primer archivo podemos apreciar como el Estacionario PMX tiene un menor promedio de coste con respecto a los demás algoritmos, y además respecto al tiempo es el que menor media de tiempo obtiene.

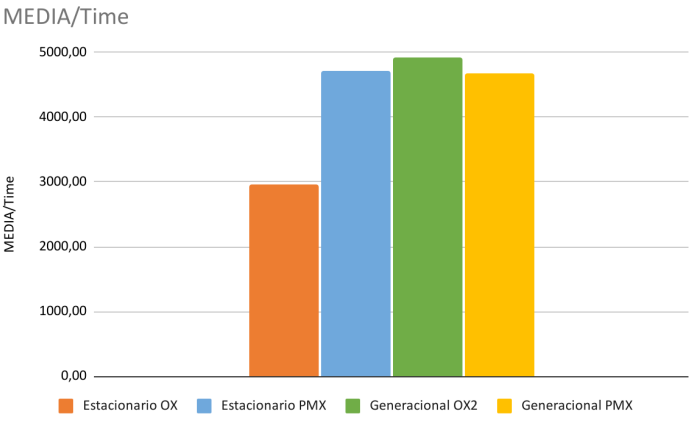
- **Nissan02:** Para el segundo algoritmo se puede observar un caso excepcional en el que todos los algoritmos obtienen un promedio de costes igual, por tanto a la hora de elegir el mejor algoritmo para este fichero, nos basaremos en función del tiempo, por lo que escogeremos el Estacionario OX que es el que tiene un menor tiempo.

- **Nissan03:** Para el tercer archivo se puede apreciar, valores de promedios de costes parecidas, obteniendo un menor valor en el Generacional OX2 y sobre el tiempo se observa que es el que mayor tiempo requiere para la ejecución. Aunque, en cuanto al menor tiempo, el estacionario OX es el que menor casi con 1000 milisegundos menos.
- **Nissan04:** En el caso del cuarto archivo, se observa que el menor promedio de costes es el Generacional OX2, aunque respecto al tiempo, este es el que de mayor tiempo requiere para obtener una solución. En sí, en general para los algoritmos, el

promedio de costes es bastante cercano y en el caso de tiempos, el estacionario OX es el que menor tiempo requiere para la ejecución. Si hubiera que escoger uno sería el estacionario OX que aunque el coste es cercano al mejor, el tiempo requerido es menor de la mitad del algoritmo con mejor coste.

En las siguientes gráficas, hemos recogido el promedio de coste y el tiempo medio de cada algoritmo para cada uno de los archivos, permitiéndonos así realizar una comparativa entre los distintos algoritmos en estos aspectos.





- **Generacional OX2 - PMX:** Dada la gráfica del coste se observa que el Generacional PMX obtiene una desviación de coste mayor que el Generacional OX2, obteniendo una diferencia de casi un 0.1 %. Respecto al tiempo media de ejecución del algoritmo, se puede observar como el generacional PMX obtiene un tiempo menor que el del Generacional OX2 teniendo una diferencia de 253 ms.

  Teniendo en cuenta el análisis que se ha realizado anteriormente el mejor algoritmo generacional es el Generacional PMX, pues la diferencia del promedio del costo es ínfimo, respecto a la diferencia en tiempo.

- **Estacionario OX - PMX:** Al observar las gráficas podemos ver que en cuanto al coste el Estacionario PMX obtiene un menor coste que el Estacionario OX con una diferencia de un 0.01% con lo que el Estacionario PMX es mejor en coste. Respecto al tiempo hay una diferencia bastante notable entre ambos, siendo una diferencia de 1754 ms menos a favor del estacionario OX.

  En conclusión basándonos en nuestro criterio, pensamos que con una diferencia de coste tan ínfima, y con una diferencia de tiempo tan notable, hemos decidido que el estacionario OX sería mejor que el PMX, ya que aunque el coste sea mejor en este último, es menor por centésimas, y un tiempo de ejecución tan alto en comparación a la diferencia de coste hace que elijamos el estacionario OX como mejor de ambos.

- **Estacionario OX - Generacional PMX:** teniendo en cuenta la gráfica de los costes se puede observar que el Estacionario OX obtiene una media de desviación de costes menor que el Generacional de PMX, siendo este de 0.05% menor. Con respecto a la media de tiempos de ejecución el Estacionario OX obtiene un tiempo de casi la mitad del que obtiene el Generacional PMX lo cual es una diferencia inmensa, siendo esta de 1708 ms.

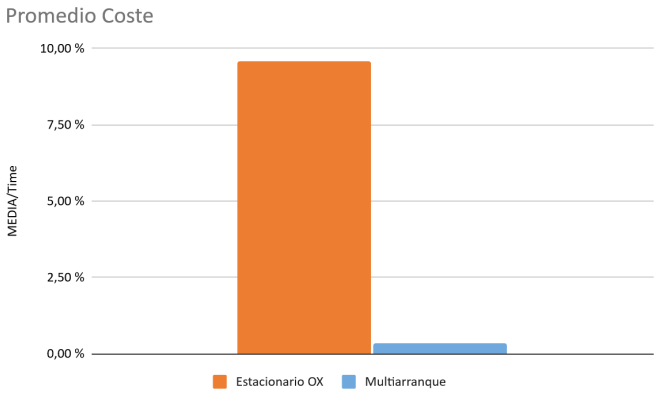
  En conclusión, siendo la diferencia de costo ínfima entre ambos algoritmos y la diferencia de tiempo tan grande, para nuestro problemas el mejor algoritmo sería el estacionario OX.

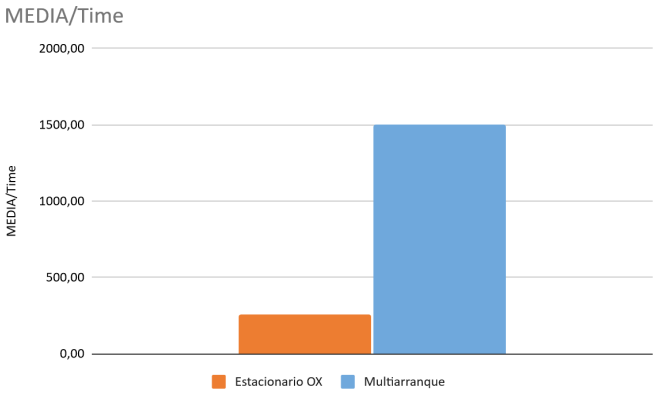
Trayectorias VS Genéticos

Aquí compararemos los algoritmos genéticos y trayectorias para problemas con datos y tamaños distintos. Compararemos los resultados obtenidos para los problemas de ford y nissan con el mejor algoritmo de trayectorias, que es el multiarranque y para el mejor algoritmo genético que hemos determinado que es el estacionario OX.

- **Problema Ford**

	Ford01		Ford02		Ford03		Ford04		MEDIA	
	C	Time	*C*	Time	*C*	Time	*C*	Time	*C*	Time
Multiarranque	**0,37 %**	933,20	**0,00 %**	1271,80	**0,95 %**	853,60	**0,00 %**	2948,80	**0,33 %**	1501,85
Estacionario OX	4,08 %	252,00	13,15 %	199,80	2,80 %	293,60	18,30 %	291,00	9,58 %	259,10



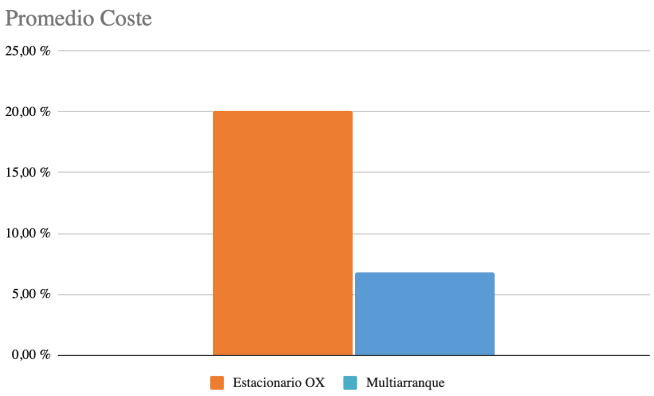


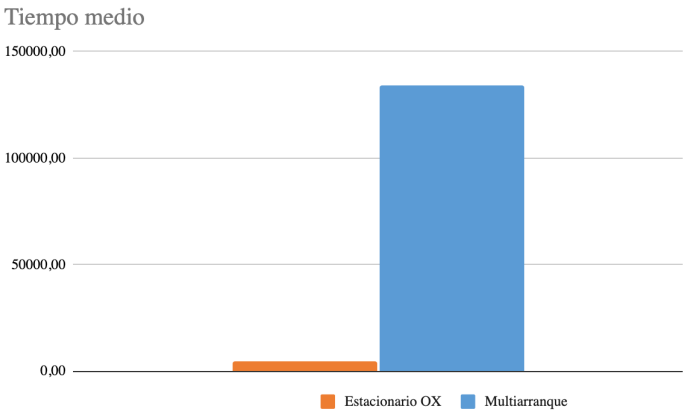
Analizando los datos anteriores de la tabla y las gráficas, se puede observar que el algoritmo de trayectorias del multiarranque funciona bastante bien para este problema, obteniendo un costo inferior en un 9.25% respecto al algoritmo genético.

Aunque cabe destacar que el tiempo necesario para calcular la solución del multiarranque es 1242 ms superior al estacionario OX.

- **Problema Nissan**

	Nissan01		Nissan02		Nissan03		Nissan04		MEDIA	
	C	Time	*C*	Time	*C*	Time	*C*	Time	*C*	Time
Multiarranque	**6,09 %**	17883,00	**7,18 %**	18784,00	**9,59 %**	54478,00	**4,18 %**	444618,00	**6,76 %**	133940,75
Estacionario OX	12,01 %	**2117,80**	32,40 %	**1828,20**	24,93 %	**3501,20**	11,03 %	**11554,20**	20,09 %	**4750,35**





Teniendo en cuenta los datos de las gráficas y las tablas para el problema de Nissan, se puede observar que el algoritmo de trayectorias funciona bastante bien, obteniendo un costo bajo en diferencia que el estacionario OX aun con la deficiencia del tiempo que requiere que es bastante elevado, siendo esta diferencia de 129190 ms que sería aproximadamente 21.5 minutos de diferencia.

CONCLUSIÓN

Podemos observar que los algoritmos de trayectorias como el multiarranque para problemas pequeños dan un buen costo con un tiempo de ejecución aceptable, mientras que para problemas cuyo tamaño es muy grande obtiene un buen costo, pero la cantidad de tiempo para el cálculo del mismo es demasiado elevada.

Por otro lado los algoritmos genéticos, no obtienen una solución muy cercana a los óptimos globales, pero mantienen la cantidad de tiempo requerida para el cálculo del mismo dentro de unos rangos razonables.
Por tanto para elegir el tipo de algoritmo a utilizar, dependerá de si necesitamos una solución muy cercana a la óptima sin importar el tiempo necesario para obtenerla, o si preferimos una solución menos precisa pero dentro de unos tiempos razonables.