Práctica 2

Genéticos

Autores:
David Díaz Jiménez 77356084T
Andrés Rojas Ortega 77382127F

Grupo 2

Metaheurísticas

Informe de prácticas

David Díaz Jiménez, Andrés Rojas Ortega

Contents

1 Definición y análisis del problema **3**
1.1 Representación de la solución . . . 3
1.2 Función objetivo . . . 3
1.3 Operadores comunes . . . 3

2 Clases auxiliares **3**
2.1 Archivo . . . 3
2.2 Configurador . . . 4
2.3 ElementoSolucion . . . 4
2.4 GestorLog . . . 4
2.5 Metaheuristicas . . . 4
2.6 Pair . . . 4
2.7 RandomP . . . 4
2.8 Timer . . . 4

3 Pseudocódigo **5**
3.1 Algoritmo principal . . . 5
3.2 Generación de la población inicial . . . 6
3.3 Operador de evaluación . . . 7
3.4 Selección de los individuos élites . . . 8
3.5 Operador de cruce . . . 9
3.6 Operador de selección . . . 10
3.7 Torneo binario . . . 10
3.8 Cruce MPX . . . 11
3.9 Cruce en dos puntos . . . 13
3.10 Operador de reparación . . . 15
3.11 Función solución . . . 16
3.12 Calculo de aportes . . . 17
3.13 Cálculo del individuo con mayor aporte . . . 18
3.14 Operador de mutación . . . 19
3.15 Operador de elitismo . . . 20

4 Experimentos y análisis de resultados **22**
4.1 Procedimiento de desarrollo de la práctica 22
4.1.1 Equipo de pruebas 22
4.1.2 Manual de usuario 22
4.2 Parámetros de los algoritmos 23
4.2.1 Genético 23
4.2.2 Semillas 23
4.3 Análisis de los resultados 23
4.3.1 Efectos de la mutación 23
4.3.2 Cruce MPX con elitismo 2 vs elitismo 3 33
4.3.3 Cruce en dos puntos con elitismo 2 vs elitismo 3 36
4.3.4 Cruce Mpx con elitismo 3 vs dos puntos con elitismo 3 . . 36

1 Definición y análisis del problema

Dado un conjunto N de tamaño n, se pide encontrar un subconjunto M de tamaño m, que maximice la función:

$$C(M) = \sum_{s_i, s_j \in M} d_{ij}$$

donde d_{ij} es la diversidad del elemento s_i respecto al elemento s_j

1.1 Representación de la solución

Para representar la solución se ha optado por el uso de un vector de enteros, en el que el elemento contenido en cada posición se corresponde con un integrante de la solución. La solución vendrá dada por las siguientes restricciones:

- La solución no puede contener elementos repetidos.
- Debe tener exactamente m elementos.
- El orden de los elementos es irrelevante.

1.2 Función objetivo

$$C(M) = \sum_{s_i, s_j \in M} d_{ij}$$

1.3 Operadores comunes

El operador de intercambio es el 1-opt, se seleccionara un elemento de la solución actual en base a un criterio y se sustituirá por un elemento que no pertenece a la solución.

2 Clases auxiliares

A continuación se enumeran las diferentes clases auxiliares utilizadas en el programa acompañadas de una breve descripción de las mismas.

nota: Para obtener información detallada se deben consultar los comentarios insertados en el código de cada una de las clases.

2.1 Archivo

Esta clase se encarga de almacenar toda la información que se encuentra dentro de cada archivo que contiene cada uno de los problemas.

2.2 Configurador

Utilizamos esta clase para leer y almacenar los parámetros del programa que se encuentran dentro del archivo de configuración.

2.3 ElementoSolucion

Clase encargada de representar a un elemento perteneciente a una solución y almacenar toda la información necesaria para la ejecución de las metaheurísticas del programa.

2.4 GestorLog

La función principal de esta clase es la administración de los archivos Log del programa y el almacenamiento de información para debug en los mismos.

2.5 Metaheuristicas

Esta clase se utiliza para lanzar la ejecución de los algoritmos para cada problema facilitado como parámetro.

2.6 Pair

Representa un par formado por un candidato y un coste asociado a este.

2.7 RandomP

Clase para generar números aleatorios.

2.8 Timer

Clase para gestionar los tiempos de ejecución del algoritmo.

3 Pseudocódigo

3.1 Algoritmo principal

Algorithm 1 Algoritmo Genético

```
poblacion ← GeneraPoblacionInicial(semilla)
Evaluacion(poblacionPadres)
while evaluacionesRelizadas<evaluacionesLimite do
  poblacionPadres ← SeleccionaPoblacion(poblacionPadres, semilla)
  poblacionHijos ← CruzarPoblacion(poblacionPadres, semilla)
  Reparar(poblacionHijos)
  MutarPoblacion(poblacionHijos,semilla)
  Evaluacion(poblacionHijos)
  poblacionPadres ← ReemplazarElite(poblacionHijos, elite)
end while
```

La primera acción que se realiza en la función principal es la generación de un conjunto de individuos aleatorios que se almacenará en la variable global "poblacion". La función encargada de realizar esta labor es "GeneraPoblacionInicial(semilla)".

A continuación, calculamos los costes de todos y cada uno de los individuos pertenecientes a "poblacionPadres". Esta tarea sera encomendada a la función "Evaluacion(poblacionPadre)".

Damos paso a la ejecución de un bucle while hasta que alcancemos el número de evaluaciones objetivo. Este número de evaluaciones lo indica la variable "evaluacionesLimite", parámetro del programa.

Lo primero que se realiza dentro del bucle es seleccionar el conjunto de individuos necesarios para realizar el cruce. Estos individuos serán almacenados en la variable "poblacionPadres", y la función encargada de realizar dicha tarea será "SeleccionaPoblacion(poblacionPadres,semilla)".

Procedemos ahora a cruzar "poblacionPadres". "CruzarPoblacion(poblacionPadres,semilla)" es la función que realiza dicha tarea, y la población resultante se guarda en la variable "poblacionHijos".

Una vez tenemos nuestra variable "poblacionHijos" con todos los individuos cruzados, debemos revisar que todos ellos cumplan con las restricciones del problema. En el caso de que algún individuo no cumpla con alguna de estas, debemos repararlo para que sea válido. Esta tarea se la asignaremos a la función "Reparar(poblacionHijos)".

Procedemos a aplicar el operador de mutación sobre "poblacionHijos" aplicando la función "MutarPoblacion(poblacionHijos,semilla)".

Volvemos a ejecutar la función "Evaluacion(poblacionHijos)" para calcular los costes de los individuos mutados.

Para finalizar, ejecutamos la función "ReemplazarElite(poblacionHijos,elite)". Esta función implementa la funcionalidad del operador de elitismo, el resultado de la ejecución la guardamos en la variable "poblacionPadres". Con este último paso estamos listos para iniciar la ejecución de una nueva iteración.

3.2 Generación de la población inicial

Entrada:

Semilla: Instancia de la clase RandomP utilizada para generar números aleatorios.

Salida:

poblacion: conjunto de individuos generados aleatoriamente.

Algorithm 2 GeneraPoblacionInicial(semilla)

```
individuo ← ∅
poblacion ← ∅
while tamañoPoblacion<numIndividuosPoblacion do
  while numGenesIndividuo<numGenesIndividuos) do
    genAleatorio ← GeneraEnteroAleatorio(semilla)
    if genAleatorio ∉ individuo then
      individuo ← individuo ∪ {genAleatorio}
    end if
  end while
  poblacion ← poblacion ∪ {individuo}
  individuo ← ∅
end while
return poblacion
```

Inicializamos las variables "individuo" y "poblacion". La variable "individuo" se utilizará como contenedor de todos los genes que se vayan generando aleatoriamente, se inicializa como un conjunto vacío. "poblacion" irá almacenando cada uno de los individuos generados, se inicializa como un conjunto vacío.

Hasta que "poblacion" no contenga el número de individuos especificado como parámetro del programa hacemos lo siguiente:

Generamos un genotipo aleatorio haciendo uso de la función "GeneraEnteroAleatorio(semilla)" y lo almacenamos en la variable local "genAleatorio".

Comprobamos que "genAleatorio" no se encuentre ya contenido dentro de "individuo" y lo añadimos en el caso de que cumpla con esta condición.

Repetimos la generación aleatoria de genotipos hasta que el número de los mismos contenido en "individuo" se corresponda con el número de genotipos pasado como parámetro del programa.

Cuando "individuo" tiene el número de genes deseado, se añade a "poblacion" y acto seguido se modifica el valor de "individuo" a vacío para dar paso a la generación de otro "individuo" nuevo.

Una vez tengamos hayamos completado "poblacion", la devolvemos como resultado de la ejecución de la función.

3.3 Operador de evaluación

Entrada:

poblacion: Conjunto de individuos de los que queremos calcular su coste.

Algorithm 3 Evaluacion(poblacion)

```
mejorCoste ← 0
for cromosoma ∈ poblacion do
  if cromosomaRecalcular! = true then
    coste ← EvaluarSolucion(cromosoma)
    cromosomaCoste ← coste
    evalucionesRealizadas ← evaluacionesRelizadas + 1
    if coste > mejorCoste then
      mejorCoste ← coste
    end if
  end if
end for
```

Antes que nada, inicializamos el valor de "mejorCoste" a 0. Esta variable local almacenará el mejor coste calculado de entre todos los individuos de la población.

Para cada cromosoma perteneciente a "poblacion" comprobamos si se tiene que calcular su coste. Esta información la contiene el atributo "recalcular" de "cromosoma".

En el caso de que debamos calcular el coste, llamamos a la función encargada de dicha tarea: "EvaluarSolucion(cromosoma)". El resultado lo almacenamos en la variable local "coste".

Cuando tengamos el coste calculado, actualizamos el parámetro "coste" de "cromosoma" con este valor.

Para finalizar, comprobamos si "coste" es mejor que "mejorCoste". Si se da esta condición, actualizamos el valor de "mejorCoste" con el de "coste".

3.4 Selección de los individuos élites

Entrada:

poblacion: Conjunto de individuos de entre los cuales queremos seleccionar el conjunto de élites.

Salidas:

individuosElites: Conjunto de los individuos élites de poblacion.

Algorithm 4 SeleccionElites(poblacion)

```
individuosElites ← ∅
mejor ← ∅
costeMejor ← 0
while individuosElites.tamaño()<numElites do
  for individuo ∈ poblacion do
    if (individuo.coste>costeMejor)∧(individuo ∉ individuosElite) then
      mejor ← individuo
      costeMejor ← individuo.coste
    end if
  end for
  individuosElites ← individuosElites ∪ {mejor}
end while
return individuosElites
```

El primer paso de todos es inicializar las variables locales "individuosElites", "mejor" y "costeMejor". "individuosElites" es un vector encargado de ir almacenando los mejores individuos de "población" que se encuentren, se inicializa como un conjunto vacío. "mejor" se utiliza para ir almacenan uno a uno los individuos de "poblacion" que presenten un mejor coste para guardarlos posteriormente en "individuosElites", su valor se inicia como vacío. "costeMejor" guarda el coste del mejor individuo encontrado hasta el momento para poder comparar con el resto de individuos de "poblacion", su valor se inicia a cero.

El algoritmo de la función realiza un bucle for hasta que el tamaño de "individuosElites" se corresponda con "numElites", es decir, el número de individuos elites que se guardan. El valor de "numElites" se deberá pasar como parámetro del programa.

El bucle for consiste en recorrer todos los individuos de "poblacion" y, si el coste de "individuo" mejora a "mejorCoste" y no ha sido introducido aún en "individuosElites", se actualizan los valores de "mejor" y "mejorCoste" con los datos del individuo. Cuando se termina el bucle for se introduce el mejor individuo encontrado en esa iteración del bucle while en "individuosElites".

Una vez que hayamos completado "individuosElites", lo devolvemos como resultado de la ejecución de la función.

3.5 Operador de cruce

Entradas:

poblacion: Conjunto de individuos que queremos cruzar.

Semilla: Instancia de la clase RandomP utilizada para generar números aleatorios.

Salida:

poblacionHijos: Conjunto de individuos resultado del cruce de poblacion.

Algorithm 5 CruzarPoblacion(poblacion,semilla)

$poblacionHijos \leftarrow \emptyset$
if tipoCruceMPX **then**
  $poblacionHijos \leftarrow RealizaCruceMPX(poblacion)$
else
  $poblacionHijos \leftarrow RelizaCruce2p(poblacion)$
end if
return poblacionHijos

Se inicializa el valor de "poblacionHijos" a un conjunto vacío para evitar errores.

Damos paso a ejecutar el cruce. El tipo de cruce viene determinado por la variable "tipoCruceMPX", un booleano.

Si el valor de "tipoCruceMPX" resulta positivo, se lanza la ejecución de "RealizaCruceMPX(poblacionHijos)". En caso contrario, se lanza "RealizaCruce2p(poblacionHijos)".

Una vez se haya realizado la ejecución del cruce, se devuelve "poblacionHijos" como resultado de ejecutar la función.

3.6 Operador de selección

Entradas:

poblacion: Conjunto de individuos del cual queremos generar la selección.

Semilla: Instancia de la clase RandomP utilizada para generar números aleatorios.

Salida:

poblacionHijos: Conjunto que contiene todos los individuos seleccionados.

Algorithm 6 SeleccionaPoblacion(poblacion,semilla)

$poblacionHijos \leftarrow \emptyset$
while tamañoPoblacionHijos<numHijos **do**
  $individuoSeleccionado \leftarrow SeleccionaIndividuo(poblacion, semilla)$
  $poblacionHijos \leftarrow poblacionHijos \cup \{individuoSeleccionado\}$
end while
return poblacionHijos

Mientras que el tamaño de "poblacionHijos" sea inferior a "numHijos" (el número de hijos de cada generación) se realizará lo siguiente:

Se selecciona por torneo binario un individuo perteneciente a "poblacion" haciendo uso de la función "SeleccionaIndividuo(poblacion,semilla)". El individuo que resulte ganador se almacena en la variable "individuoSeleccionado".

"individuoSeleccionado" se añade a "poblacionHijos" y se repite el proceso.

Una vez "poblacionHijos" alcanza el tamaño deseado, finaliza la ejecución de la función y se devuelve como resultado.

3.7 Torneo binario

Entradas:

poblacion: Conjunto de individuos.

Semilla: Instancia de la clase RandomP utilizada para generar números aleatorios.

Salida:

Devuelve el individuo seleccionado de poblacion.

Algorithm 7 SeleccionaIndividuo(poblacion,semilla)

```
seleccionado1 ← GeneraEnteroAleatorio(semilla)
seleccionado2 ← GeneraEnteroAleatorio(semilla)
while seleccionado1==seleccionado2 do
  seleccionado2 ← GeneraEnteroAleatorio(semilla)
end while
if poblacion[seleccionado1].coste > poblacion[seleccionado2].coste then
  return poblacion[seleccionado1]
else
  return poblacion[seleccionado2]
end if
```

Para realizar el torneo binario lo primero que necesitamos es generar dos números aleatorios que se corresponderan con los índices en los que se encuentran los individuos seleecionados. Realizaremos tal generación de números aleatorios con la función "GeneraEnteroAleatorio(semilla)", y almacenaremos los valores resultantes en las variables "seleccionado1" y "seleccionado2".

Comprobamos que no se repitan los valores de "seleccionado1" y "seleccionado2". Si ocurre esto, generamos otro nuevo valor para "seleccionado2" hasta que obtengamos uno válido.

Para finalizar, comparamos el coste de los dos individuos seleccionados y devolvemos como resultado aquel que posea un mejor coste.

3.8 Cruce MPX

Entrada:

poblacionPadres: conjunto de individuos sobre la cual se realizará el cruce.

Salida:

poblacionHijos: Conjunto de individuos resultado del cruce de poblacion.

Algorithm 8 RealizarCruceMPX(poblacionPadres)

```
for i=0; i<tamañoPoblación & TamañoPoblacionHijos < tamañoPoblacion;
i+=2 do
  probRepro ← GeneraFloatAleatorio(semilla)
  padre1 ← i
  padre2 ← i + 1
  if probRepro < probabilidadReproduccion then
    hijo ← ∅
    for gen in padre1 do
      prob ← GeneraFloatAleatorio(semilla)
      if prob < probabilidadMPX then
        hijo ← hijo ∪ gen
      end if
    end for
    for gen in padre2 do
      hijo ← hijo ∪ gen
    end for
    poblacionHijos ← poblacionHijos ∪ {hijo}
  else
    poblacionHijos ← poblacionHijos ∪ {padre1}
    if TamañoPoblacionHijos < tamañoPoblación then
      poblacionHijos ← poblacionHijos ∪ {padre2}
    end if
  end if
  if i == tamañoPoblacion -2 then
    i ← 0
  end if
end for
return poblacionHijos
```

Iniciamos un bucle for. Este bucle for se ejecutará hasta que "poblacionHijos" tenga un tamaño "numIndividuosPoblacion" (parámetro del programa).

Almacenamos el primer par de padres, obtenido de la variable "poblacionPadres", en las variables "padre1" y "padre2". Además, generamos un número float aleatorio con "GeneraFloatAleatorio(semilla)" y lo almacenamos en la variable "probRepro".

Comprobamos si la pareja de padres debe reproducirse. Esto lo conseguimos comparando "probRepro" con la variable "probabilidadCruce" (parámetro del programa).

En el caso de que se deban reproducir, inicializamos una variable llamada "hijo" al valor conjunto vacío. En esta variable almacenaremos el material genético del hijo resultado de cruzar "padre1" y "padre2".

Recorremos todos los genes de "padre1" y comprobamos para cada gen si se debe incluir en el material genético de "hijo" o no. Esto lo comprobamos generando un número float aleatorio con "GeneraFloatAleatorio(semilla)" y comparando el resultado (almacenado en "prob") con "probabilidadMPX".

En el caso de que un gen de "padre1" se incluya en "hijo", eliminamos el gen de "padre2" (en el caso de que este exista en "padre2").

Cuando hayamos terminado de incluir material genético de "padre1" en "hijo", procedemos a añadir todos los genes de "padre2" en "hijo".

Para finalizar, añadimos el hijo generado a "poblacionHijos" y pasamos al siguiente par de padres.

En el caso de que el par de padres no se deba reproducir, añadimos directamente cada padre a "poblacionHijos" siempre que el tamaño de "poblacionHijos" no haya llegado a "numIndividuosPoblacion".

Antes de iniciar una nueva iteración del bucle for, comprobamos que "i" no haya rebasado "numIndividuosPoblacion". Si este es el caso, reiniciamos el valor de "i" a 0. Realizamos esto para no sobrepasar el tamaño de "poblacionPadres".

Una vez terminamos de ejecutar el bucle for principal, devolvemos "poblacionHijos" como resultado de la ejecución de la función.

3.9 Cruce en dos puntos

Entrada:

poblacionPadres: conjunto de individuos sobre la cual se realizará el cruce.

Salida:

poblacionHijos: Conjunto de individuos resultado del cruce de poblacion.

Algorithm 9 RealizarCruce2p(poblacionPadres)

```
for i=0; i<tamañoPoblación; i+=2 do
  probRepro ← GeneraFloatAleatorio(semilla)
  if probRepro < probabilidadReproducción then
    corte1 ← GeneraEnteroAleatorio(semilla)
    corte2 ← GeneraEnteroAleatorio(semilla)
    padre1 ← poblacionPadres[i]
    padre2 ← poblacionPadres[i + 1]
    while corte1==corte2 do
      corte2 ← GeneraEnteroAleatorio(semilla)
    end while
    for j=0;j<corte1;j++ do
      hijo1 ← hijo1 ∪ padre1.getGen[j]
      hijo2 ← hijo2 ∪ padre2.getGen[j]
    end for
    for j=corte1;j<corte2;j++ do
      hijo1 ← hijo1 ∪ padre2.getGen[j]
      hijo2 ← hijo2 ∪ padre1.getGen[j]
    end for
    for j=corte2;j<tamañoIndividuo;j++ do
      hijo1 ← hijo1 ∪ padre1.getGen[j]
      hijo2 ← hijo2 ∪ padre2.getGen[j]
    end for
    poblacionHijos ← poblacionHijos ∪ {hijo1}
    poblacionHijos ← poblacionHijos ∪ {hijo2}
  else
    poblacionHijos ← poblacionHijos ∪ {poblacionPadres[i]}
    poblacionHijos ← poblacionHijos ∪ {poblacionPadres[i + 1]}
  end if
end for
return poblacionHijos
```

Recorremos "poblacion" de dos en dos realizando lo que a continuación se expone.

Lo primero a realizar es comprobar si debemos realizar el cruce entre los dos primeros padres. Esta comprobación se realiza comparando "aleatorioCruce" y "probabilidadCruce". "aleatorioCruce" almacena un float aleatorio generado haciendo uso de la función "GeneraFloatAleatorio(semilla)"."probabilidadCruce" tiene almacenada la probabilidad de que dos individuos se reproduzcan, esta información se pasa al programa como parámetro.

En el caso de que sí se tengan que cruzar, generamos dos puntos de corte aleatorios haciendo uso de "GeneraEnteroAleatorio(semilla)" y los almacenamos

en las variables "corte1" y "corte2". Almacenamos en las variables "padre1" y "padre2" los individuos a cruzar.

Comprobamos que los cortes generados no sean los mismos y, si lo son, generamos otro valor aleatorio para "corte2" hasta que los dos cortes dejen de ser iguales.

Rellenamos "hijo1" con lo genotipos de "padre2" e "hijo2" con los genotipos de "padre1" hasta llegar a "corte1". Rellenamos los genotipos a continuación de "corte1" de "hijo1" con los genotipos de "padre1", y los genotipos de "hijo2" con los de "padre2" hasta llegar a "corte2". A partir de "corte2" y hasta llegar al final, rellenamos "hijo1" con los genotipos de "padre2", e "hijo2" con los genotipos de "padre1".

Cuando hayamos completado el cruce, sobreescribimos el valor de la posición de "padre1" y "padre2" con el valor de "hijo1" e "hijo2" dentro de "poblacionHijos".

Una vez hayamos terminado el bucle for principal, tendremos almacenados en "poblacionHijos" todos los nuevos individuos resultantes del cruce. Devolvemos "poblacionHijos" como resultado.

3.10 Operador de reparación

Entrada:

poblacionHijos: Conjunto de individuos que queremos reparar.

Algorithm 10 Reparar(poblacionHijos)

for $individuo \in poblacionHijos$ **do**
  if !FuncionSolucion(individuo) **then**
    if tamañoIndividuo>tamañoIndividuoProblema **then**
      while tamañoIndividuo>tamañoIndividuoProblema **do**
        $elementoMenor \leftarrow CalcularAportes(individuo)$
        $individuo \leftarrow individuo - \{elementoMenor\}$
      end while
    else if tamañoIndividuo<tamañoIndividuoProblema **then**
      $elementoMayor \leftarrow CalcularMayorAporte(individuo)$
      $individuo \leftarrow individuo \cup \{elementoMayor\}$
    end if
  end if
end for

Para cada individuo perteneciente a "poblacionHijos" se comprueba que se cumple con todas las restricciones. Esta labor recae sobre la función "FuncionSolucion(individuo)".Si se cumplen todas las restricciones no se hace nada y se pasa al siguiente individuo.

En el caso de que "FuncionSolucion" nos indique que no se cumple con todas las reestricciones, se comprueba cuál de ellas no se cumple para poder proceder a repararlas. Nuestras reestricciones son: el número de genes debe ser igual a "tamañoIndividuoProblema", no se debe repetir ningún gen en el individuo.

En el caso de que el número de genes del individuo sea inferior a "tamañoIndividuoProblema", vamos seleccionando el gen que mayor coste aporte al individuo con la función "CalcularMayorCoste(individuo)" y lo añadimos a los genes del individuo hasta que alcancemos "tamañoIndividuoProblema".

En el caso de que sobrepasemos "tamañoIndividuoProblema", seleccionamos el gen del individuo que menos coste aporte con la función "CalcularAportes(individuo)" y lo eliminamos. Repetimos este proceso hasta que alcancemos "tamañoIndividuoProblema".

No comprobamos que los genes no se repitan ya que en el algoritmo implementado utilizamos un Set, y esta estructura de datos resuelve este problema. En el caso de que debamos implementarlo, se eliminarían todos los elementos repetidos y, a continuación, se irían rellenando los espacios vacantes con los elementos que devuelva la función "CalcularMayorAporte" hasta que el número de genes del individuo alcance "tamañoIndividuoProblema".

Cuando hayamos realizado todas las reparaciones sobre todos los individuos pertenecientes a "poblacionHijos" ya tendremos el conjunto de individuos reparado y se finaliza la ejecución de la función.

3.11 Función solución

Entrada:

individuo: conjunto de genes que queremos comprobar si es solución.

Salida:

Devuelve un booleano indicando si es o no una solución válida.

Algorithm 11 FuncionSolucion(individuo)

```
for i=0;i<numGenesIndividuo-1;i++ do
  for j=i+1;numGenesIndividuo;j++ do
    if individuo[i]==individuo[j] then
      seRepite ← true
    end if
  end for
  numGenes ← numGenes + 1
end for
if numGenes!=numGenesIndividuo then
  malTamaño ← true
end if
return !(malTamaño ∨ seRepite)
```

El funcionamiento de esta función resulta trivial; comprueba que no se repitan entre sí los genes del individuo y que la cantidad de estos se corresponda con "tamañoIndividuoProblema".

Comparamos todos los genes de "individuo" y almacenamos en la variable "seRepite" si encontramos algún par que se repita. A la vez que comparamos los genes, vamos registrando el número de genes en la variable "numGenes".

Cuando hayamos terminado de comparar todos los genes, comprobamos que "numGenes" se corresponda con "tamañoIndividuoProblema" y almacenamos el resultado de la comprobación en la variable "malTamaño".

Para finalizar devolvemos si no se cumple alguna de las restricciones como resultado.

3.12 Calculo de aportes

Entrada:

individuo: Conjunto de genes de entre los cuales queremos encontrar el de menor aporte.

Salida:

Devuelve el gen que menos aporta a la solución.

Algorithm 12 CalcularAportes(individuo)

```
aporte ← 0
listaAportes ← ∅
for gen1 ∈ individuo do
  for gen2 ∈ individuo do
    aporte ← aporte + matrizDistancias[gen1][gen2]
  end for
  AñadirAporte(gen1,aporte)
end for
Sort(listaAportes)
return listaAportes[0]
```

El primer paso consiste en inicializar las variables globales "aporte" y "listaAportes"."aporte" almacena, como su nombre indica, el coste que se aporta a la solución; su valor inicial es cero. "listaAportes" almacena todos los genes con su respectivo aporte, su valor inicial es el conjunto vacío.

Para cada gen de "individuo" vamos recorriendo el resto de genes y sumando sus distancias a la variable "aporte".

Cuando hayamos terminado de calcular el aporte para un gen, añadimos a "listaAportes" el gen con su aporte haciendo uso de la función "AñadirAporte".

Cuando hayamos calculado todos los aportes, ordenamos "listaAportes" con la función "Sort".

Para finalizar, devolvemos el elemento que se encuentra en primera posición, que se corresponderá con el gen que menos aporta en "individuo".

3.13 Cálculo del individuo con mayor aporte

Entrada:

individuo: Conjunto de genes para el cual queremos encontrar el de mayor aporte.

Salida:

Devuelve el gen que más aportaría a la solución.

Algorithm 13 CalcularMayorAporte(individuo)

```
aporte ← 0
listaAportes ← ∅
for gen1 ∈ matrizDatos do
  for gen2 ∈ matrizDatos do
    aporte ← aporte + matrizDistancias[gen1][gen2]
  end for
  AñadirAporte(gen1,aporte)
end for
Sort(listaAportes)
return listaAportes[numGenesIndividuos-1]
```

El funcionamiento de esta función es similar a "CalcularAportes(individuo)", difiriendo en dos puntos:

Los aportes que se calculan es este caso hacen referencia a todos los genes que NO se encuentran en "individuo".

Cuando tengamos almacenados todos los genes con su aporte en "listaAportes", en este caso se devuelve el elemento que se encuentra en la última posición. Esto es debido a que queremos obtener el mejor gen que se puede incluir en "individuo" para maximizar el coste resultante.

3.14 Operador de mutación

Entradas:

poblacionHijos: Conjunto de individuos sobre el que queremos aplicar el operador de mutación.

Semilla: Instancia de la clase RandomP utilizada para generar números aleatorios.

Algorithm 14 Mutar(poblacionHijos,semilla)

```
for individuo ∈ poblacionHijos do
  elatorioMutacion ← GeneraFloatAleatorio(semilla)
  if aleatorioMutacion<=probabilidadMutacion then
    posMuta ← GeneraEnteroAleatorio(semilla)
    eleMutado ← GeneraEnteroAleatorio(semilla)
    Intercambia(individuo,posMuta,eleMutado)
  end if
end for
```

Para cada individuo perteneciente a "poblacionHijos" realizamos lo que a continuación se expone.

Se comprueba si debemos realizar la mutación comparando "aleatorioMutacion" con "probabilidadMutacion"."aleatorioMutacion" es generado aleatoriamente con la función "GeneraFloatAleatorio(semilla)". "probabilidadMutacion" contiene el parámetro del problema que indica qué probabilidad existe de que un individuo mute.

En el caso de que "individuo" deba mutar, se genera aleatoriamente tanto la posición del gen que muta como el valor por el que se va a cambiar dicho gen. La generación aleatoria se realiza haciendo uso de la función "GeneraEnteroAleatorio(semilla)".

Una vez tenemos los datos necesarios para realizar la mutación, hacemos la modificación del cromosoma con la función "Intercambia(individuo,posMuta,eleMutado)".

Cuando terminemos el bucle for principal tendremos nuestra "poblacionHijos" mutada.

3.15 Operador de elitismo

Entradas:

poblacionHijos: Conjunto de individuos sobre el que queremos aplicar el operador de elitismo.

elites: Conjunto que contiene los individuos élites.

Salida:

poblacionHijos: Conjunto de individuos resultado de aplicar el operador de elitismo.

Algorithm 15 ReemplazarElite(poblacionHijos,elites)

```
elites ← SeleccionElites(poblacionPadres)
Sort(poblacionHijos)
indice ← 0
for elite ∈ elites do
  poblacionHijos[indice] ← elite
  indice ← indice + 1
end for
return poblacionHijos
```

Antes que nada seleccionamos los individuos élites de "poblacionPadres", y a continuación ordenamos "poblacionHijos" de menor a mayor haciendo uso de la función "Sort".

Inicializamos el valor de la variable local "indice" a cero. Esta variable representa un apuntador a los elementos de la variable "poblacionHijos".

A continuación vamos almacenando cada uno de los individuo élites almacenados en la variable "elites" en posiciones inciales de "poblacionHijos", apuntadas por "indice". De este modo conseguimos eliminar los individuos con peor coste y sustituirlos por los individuos élites de la población de los padres.

Para finalizar, se devuelve "poblacionHijos" como resultado de la ejecución.

4 Experimentos y análisis de resultados

4.1 Procedimiento de desarrollo de la práctica

Para realizar la práctica, se ha optado por implementar las heurísticas propuestas en el lenguaje de programación Java. El ejecutable que se entrega junto a este documento ha sido compilado bajo Apache NetBeansIDE 12.0.

4.1.1 Equipo de pruebas

Los resultados de las heurísticas han sido obtenidos en el siguiente equipo:

- Host: 80WK Lenovo Y520-15IKBN
- S.O: KDE neon User Edition 5.20 x86 64
- Kernel: 5.4.0-52-generic
- CPU: Intel i5-7300HQ (4) @ 3.500GHz
- GPU: NVIDIA GeForce GTX 1050 Mobile
- GPU: Intel HD Graphics 630
- Memoria RAM : 7837 MiB.

4.1.2 Manual de usuario

Para ejecutar el software asegúrese de que el archivo .jar proporcionado se ubica en el mismo directorio que la carpeta *archivos*.

Cuando se muestre la GUI, podrá seleccionar la heurística que desee mediante el botón correspondiente. Una vez empiece la ejecución de una heurística no sera posible seleccionar otra hasta que finalice su ejecución. Los resultados finales se mostrarán en el cuadro de texto, a su vez, se generan los log correspondientes a cada archivo y semilla en la carpeta Log.

Figure 1: GUI

4.2 Parámetros de los algoritmos

4.2.1 Genético

Para regular el comportamiento de los algoritmos genéticos, se han definido los siguientes parámetros en el archivo de configuración:ç

- Semilla: Número utilizado para la generación de números pseudoaleatorios, valor por defecto : 77356084
- Evaluaciones: Número máximo de evaluaciones, valor por defecto: 50000
- Elitismo: Número máximo de los mejores individuos de la generación anterior que pasan a la siguiente.
- OperadorMPX: Booleano que indica si se debe usar el operador de cruce MPX (true), o el cruce en dos puntos (false).
- Probabilidad de mutacion: Probabilidad de mutación que se aplica a cada gen, valor por defecto: 0,01.
- Probabilidad de reproduccion: Probabilidad de que dos individuos de la población se crucen, valor por defecto: 0,90.
- Cromosomas: Tamaño de la población, valor por defecto: 50.
- Probabilidad MPX: Determina la cantidad de genes que se heredan del primer padre, para el operador de cruce MPX, valor por defecto: 0,50.

4.2.2 Semillas

Para la generación de números pseudoaleatorios se utiliza una semilla previamente definida en el archivo de configuración, en este caso es 77356084. Esta semilla se va rotando en las 5 iteraciones de cada archivo.

77356084 $\rightarrow$ 73560847 $\rightarrow$ 35608477 ...

4.3 Análisis de los resultados

4.3.1 Efectos de la mutación

La mutación en los algoritmos genéticos es una técnica de diversificación/exploración, la cual puede ayudar a salir de óptimos locales, no obstante, su porcentaje de aplicación debe ser reducido. Esto es debido a que una alta probabilidad de mutación puede alterar considerablemente las soluciones obtenidas, resultando en una calidad inferior de las mismas.

Para reflejar este comportamiento de la mutación, se ha seleccionado el conjunto de datos SOM, y se ha establecido 3 valores distintos de probabilidad de mutación: 0,01, 0,05 y 0,09. Cada Probabilidad de mutación se ha aplicado en 5 iteraciones, con distinta semilla para cada archivo. Posteriormente se

ha obtenido la media de las 5 iteraciones para cada archivo y probabilidad de mutación, de dichos resultados se ha obtenido la siguiente gráfica:

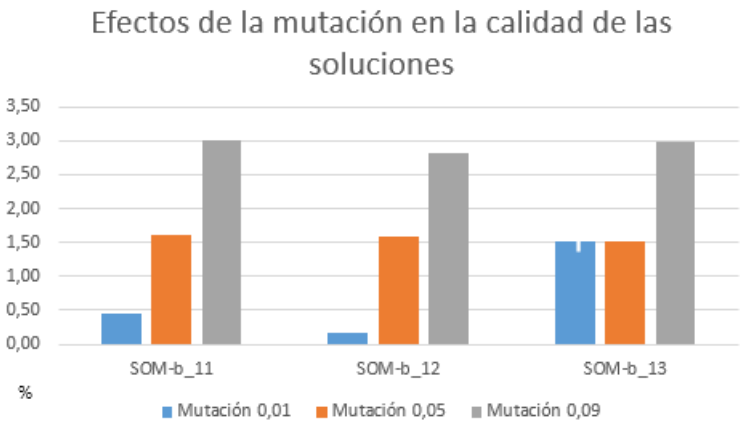
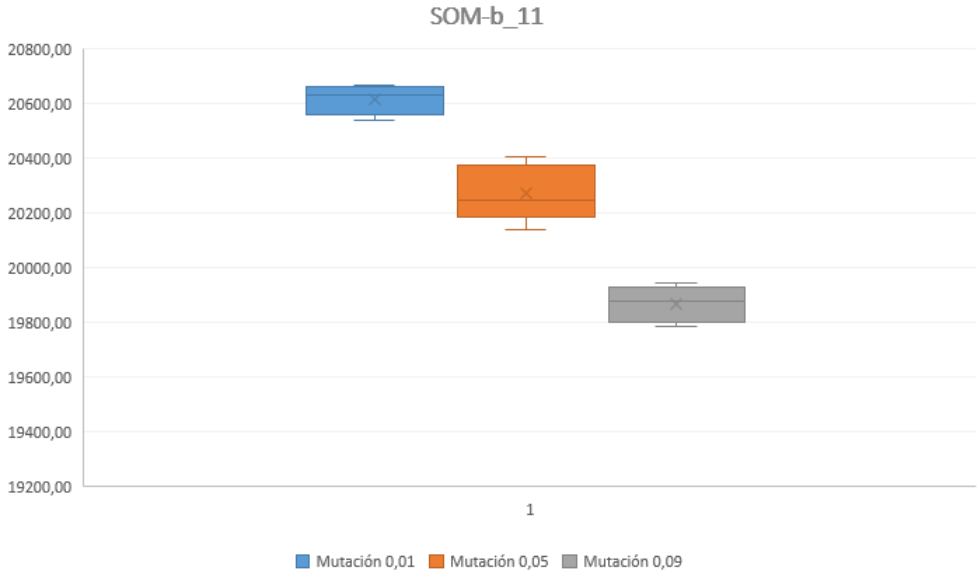
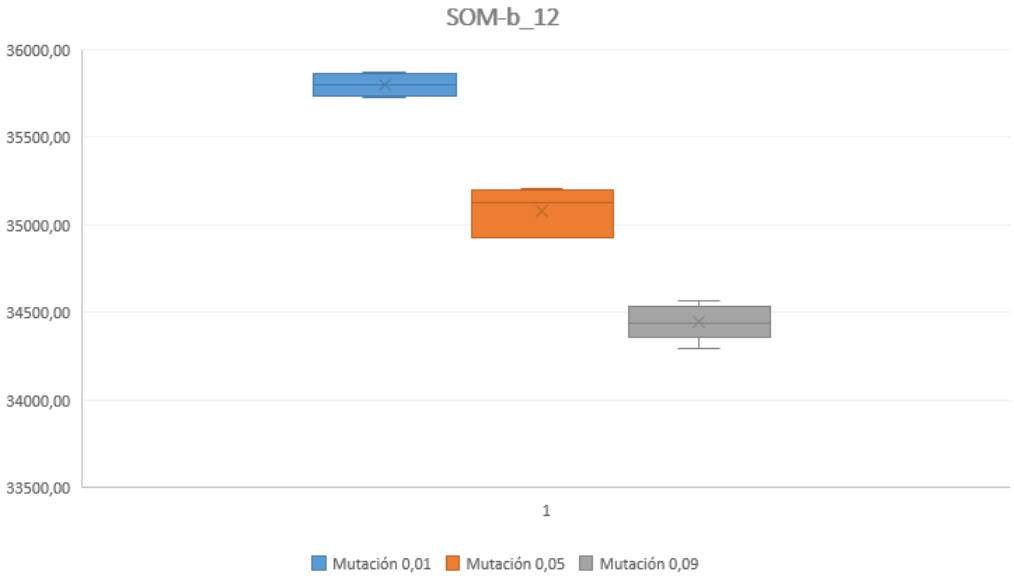
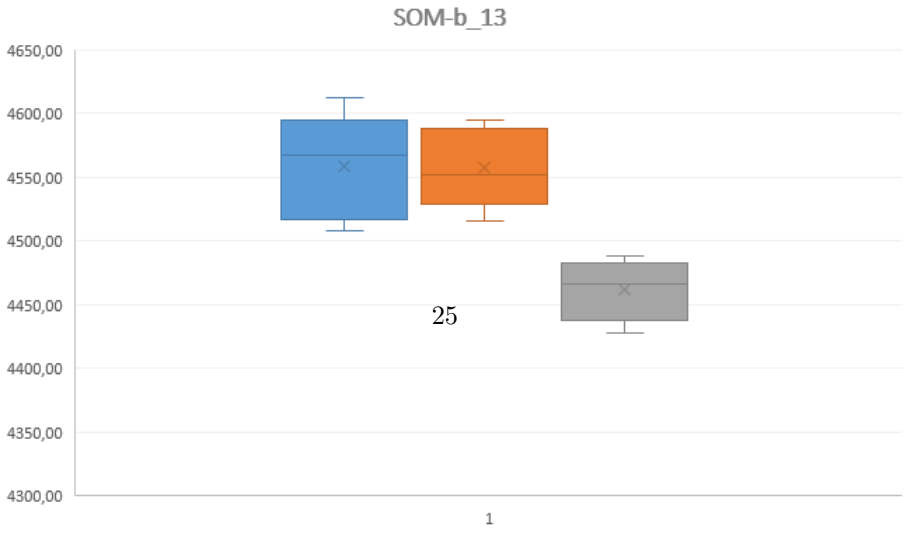


Figure 2: Diferencias en porcentajes respecto a los óptimos globales

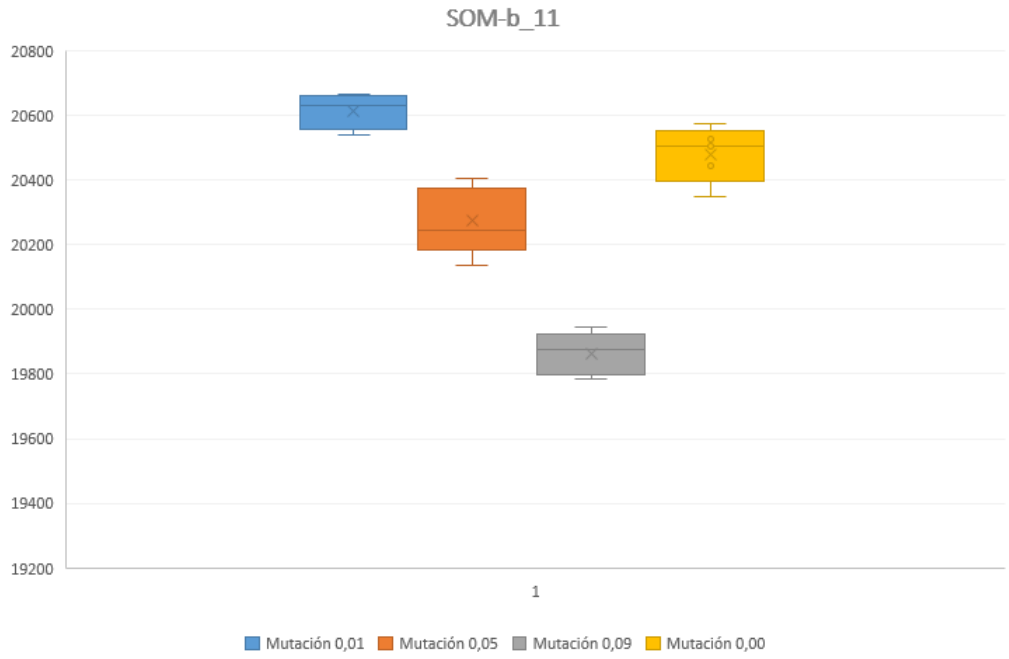
Como se puede observar, un mayor porcentaje de mutación, empeora los resultados obtenidos mediante algoritmos genéticos.

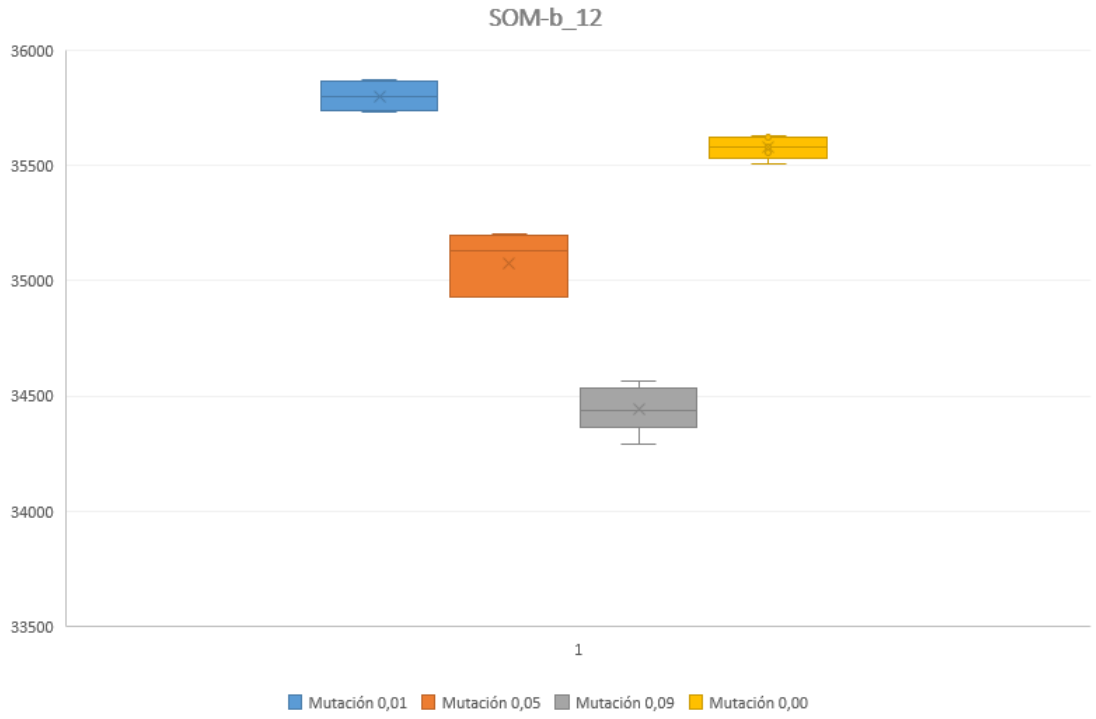

SOM-b_11
20800,00
20600,00
20400,00
20200,00
20000,00
19800,00
19600,00
19400,00
19200,00
1
Mutación 0,01
Mutación 0,05
Mutación 0,09

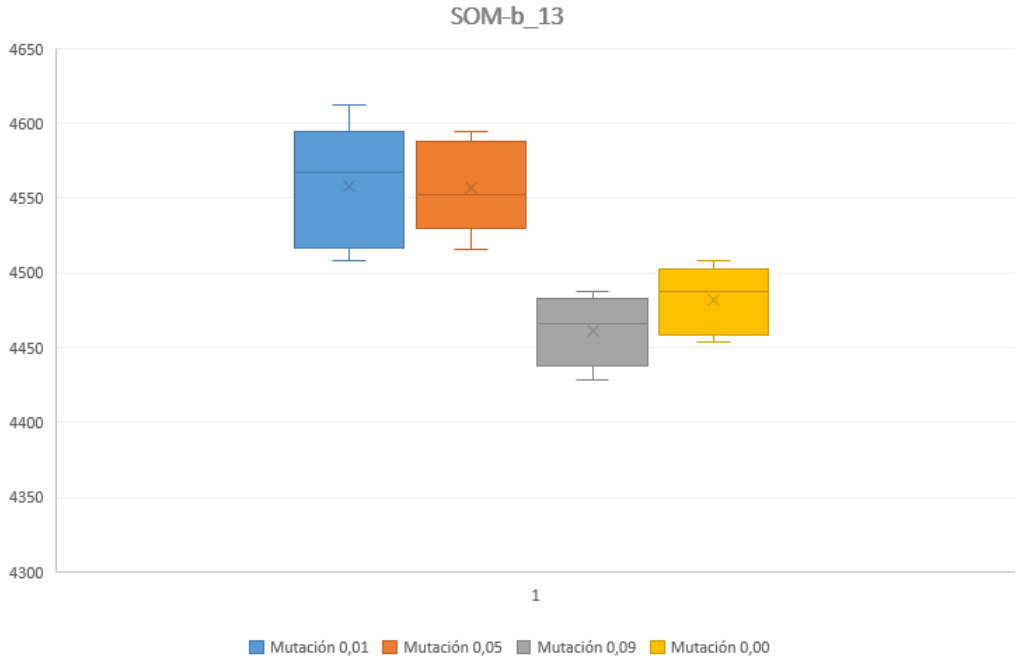

SOM-b_12
36000,00
35500,00
35000,00
34500,00
34000,00
33500,00
1
Mutación 0,01
Mutación 0,05
Mutación 0,09


SOM-b_13
4650,00
4600,00
4550,00
4500,00
4450,00
4400,00
4350,00
4300,00
1

A raíz de los resultados obtenidos anteriormente, se reflexionó sobre el impacto que tendría fijar una capacidad de mutación nula. Al realizar las pruebas, se obtuvo que fijar una mutación al 0,00 elimina parte de la capacidad de mejora, siendo sus resultados inferiores a una mutación del 0,01. Esto es debido a que el comportamiento del algoritmo sin la mutación se basa en la explotación.




SOM-b_12
36000
35500
35000
34500
34000
33500
1
Mutación 0,01
Mutación 0,05
Mutación 0,09
Mutación 0,00



En las tres gráficas siguientes se muestra la evolución del mejor coste encontrado hasta el momento en la ejecución de cada uno de los algoritmos conforme aumenta el número de evaluaciones realizadas para los problemas GKD-c1, GKD-c2, GKD-c3. La semilla utilizada ha sido 35608477.

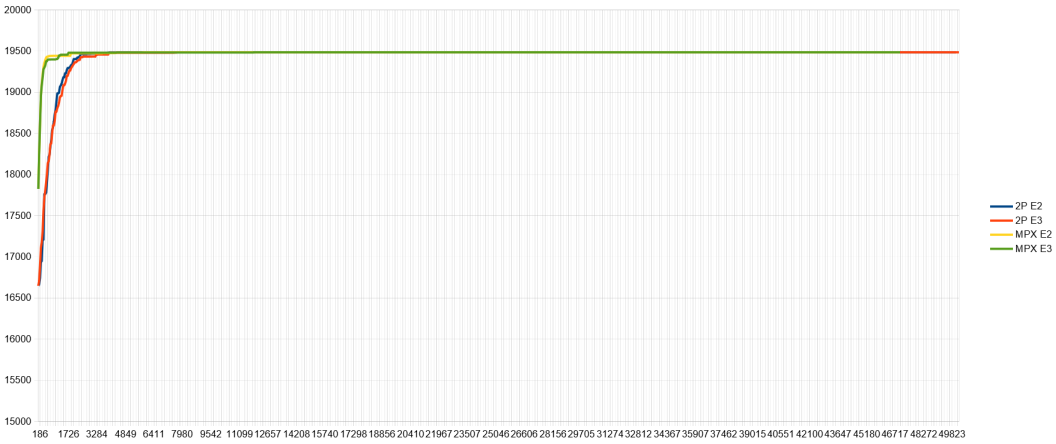


Figure 3: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema GKD-c1, semilla 35608477

Para el problema GKD-c1 los algoritmos MPX empiezan a encontrar soluciones aceptables en torno a las 100 evaluaciones realizadas para los dos elitismos. Los algoritmos de cruce en dos puntos empiezan a obtener soluciones aceptables a partir de las 420 evaluaciones para una élite = 2, y a partir de las 372 evaluaciones para una élite = 3. Hemos considerado una solución aceptable aquella solución que difiere como máximo un 10% respecto al óptimo global. En este problema las soluciones óptimas son todas aquellas cuyo coste es superior a 17.536,669.

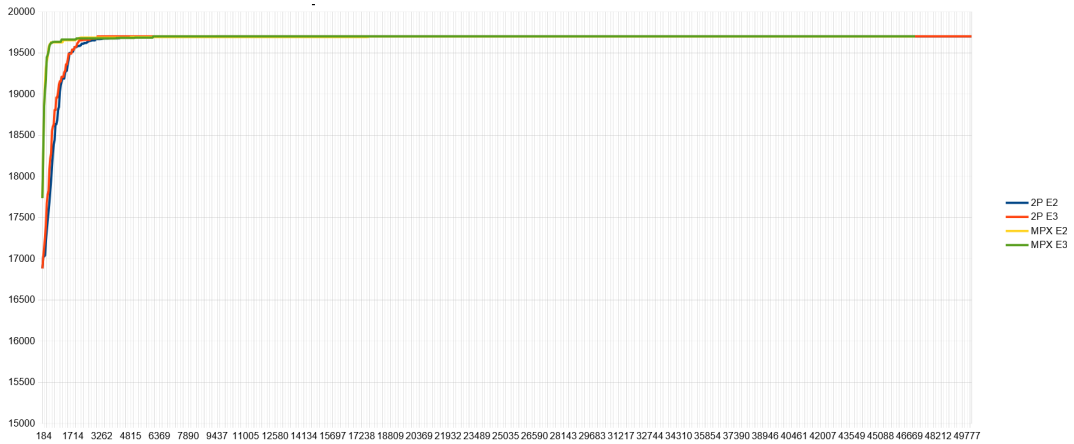


Figure 4: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema GKD-c2, semilla 35608477

Para el problema GKD-c2 los algoritmos MPX empiezan a encontrar soluciones aceptables en torno a las 100 evaluaciones realizadas para los dos elitismos. Los algoritmos de cruce en dos puntos empiezan a obtener soluciones aceptables a partir de las 321 evaluaciones para una élite = 2, y a partir de las 277 evaluaciones para una élite = 3. En este problema las soluciones óptimas son todas aquellas cuyo coste es superior a 17.731,383.

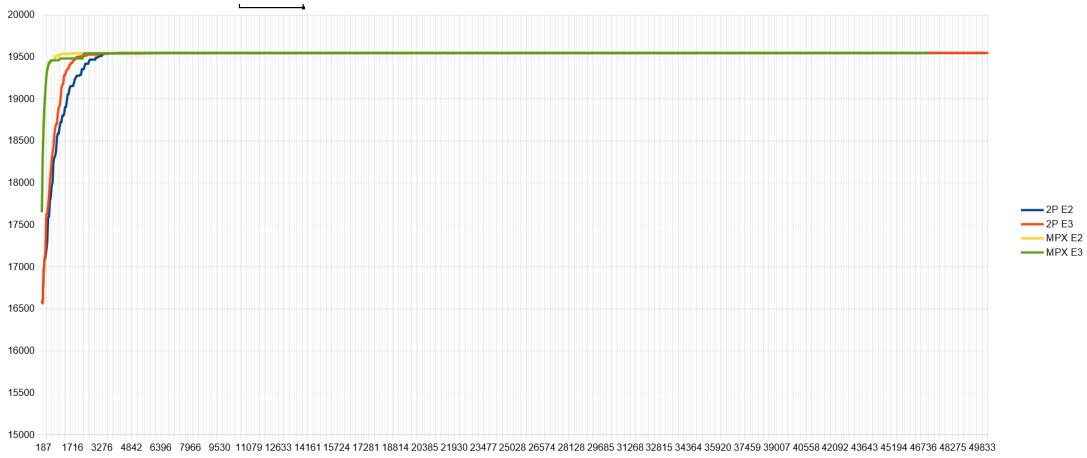


Figure 5: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema GKD-c3, semilla 35608477

Para el problema GKD-c3 los algoritmos MPX empiezan a encontrar soluciones aceptables en torno a las 100 evaluaciones realizadas para los dos elitismos. Los algoritmos de cruce en dos puntos empiezan a obtener soluciones aceptables a partir de las 467 evaluaciones para una élite = 2, y a partir de las 324 evaluaciones para una élite = 3. En este problema las soluciones óptimas son todas aquellas cuyo coste es superior a 17.592,486.

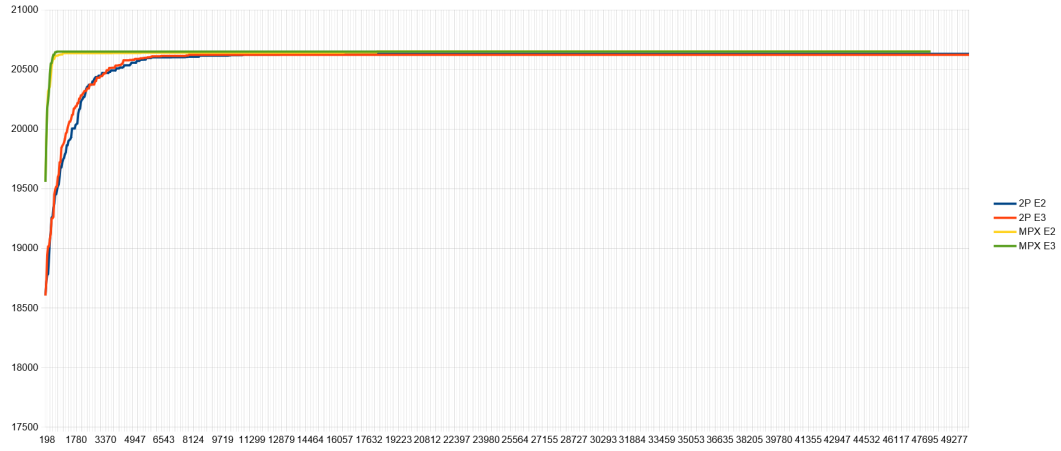


Figure 6: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema SOM-b11, semilla 35608477

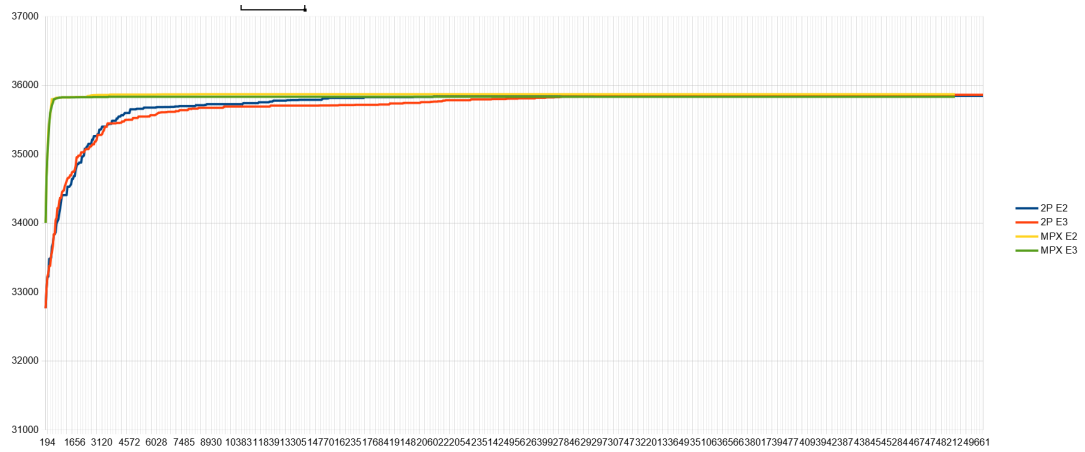


Figure 7: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema SOM-b12, semilla 35608477

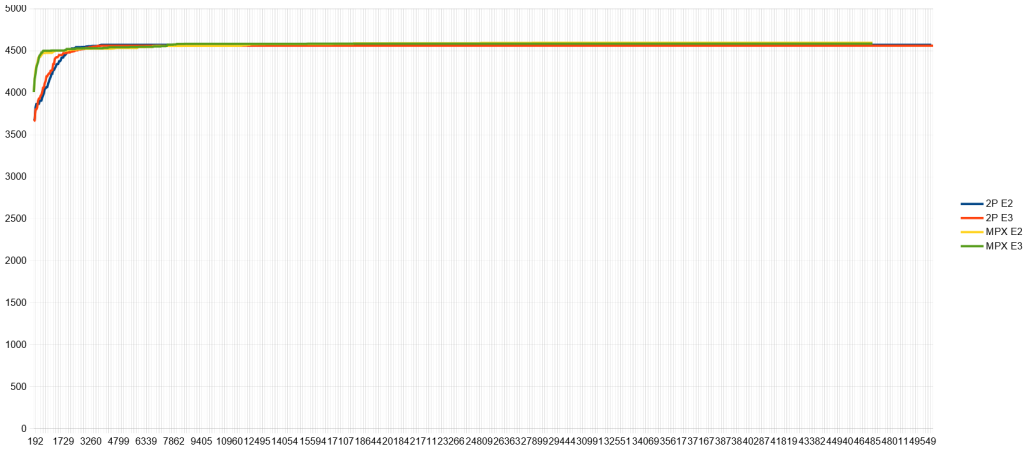


Figure 8: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema SOM-b13, semilla 35608477

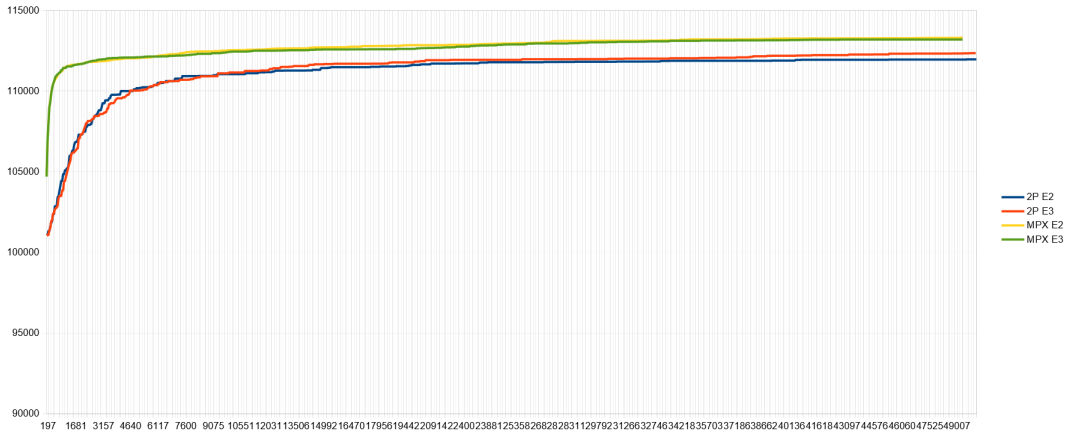


Figure 9: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema MDG-a21, semilla 35608477

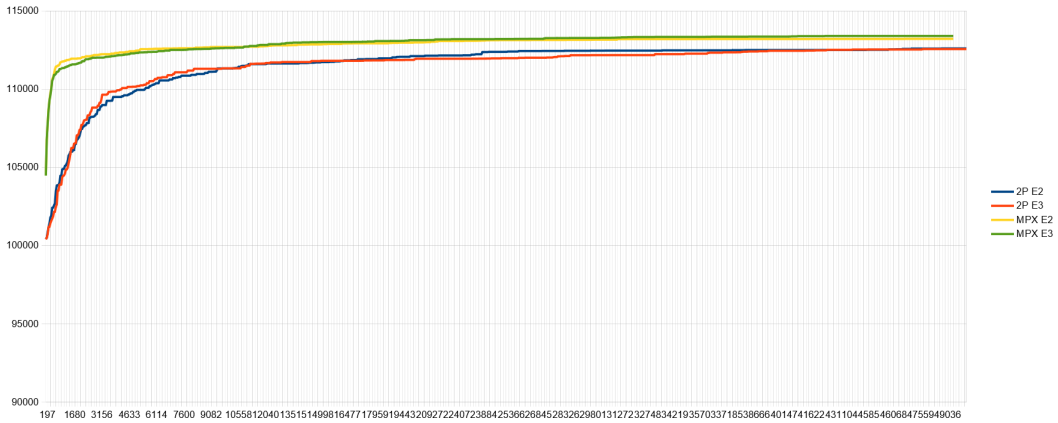


Figure 10: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema MDG-a22, semilla 35608477

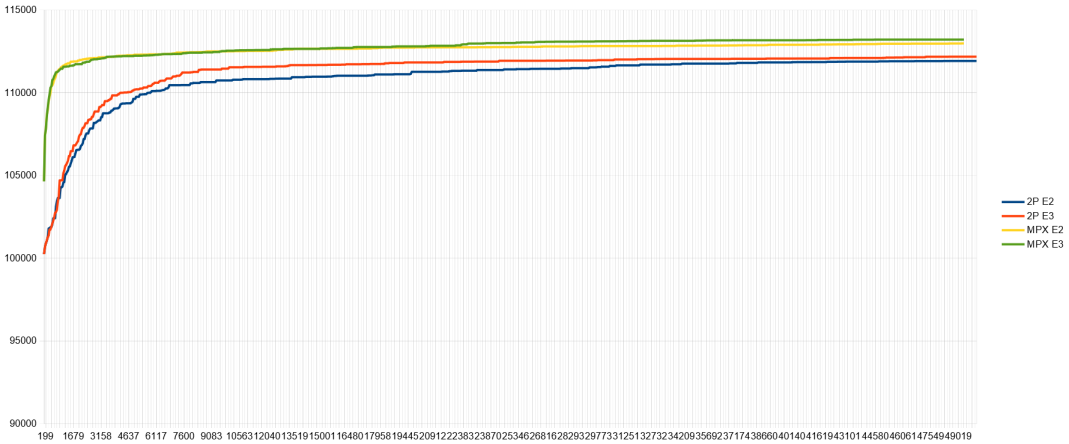


Figure 11: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema MDG-a23, semilla 35608477

4.3.2 Cruce MPX con elitismo 2 vs elitismo 3

El operador de cruce MPX aplicado a la serie de datos GKD, con una élite de 2 o 3 individuos, ofrece unos resultados muy similares respecto a coste.

Para el archivo GKD-c1 (12), los resultados obtenidos en las diferentes iteraciones, son similares, ofreciendo un ligero mayor agrupamiento de los resultados a favor de la élite de 3 individuos.

Para el archivo GKD-c2 (13),los resultados son ligeramente superiores con una élite de 3 individuos.

Para el archivo GKD-c3 (14), los resultados revelan un comportamiento similar.

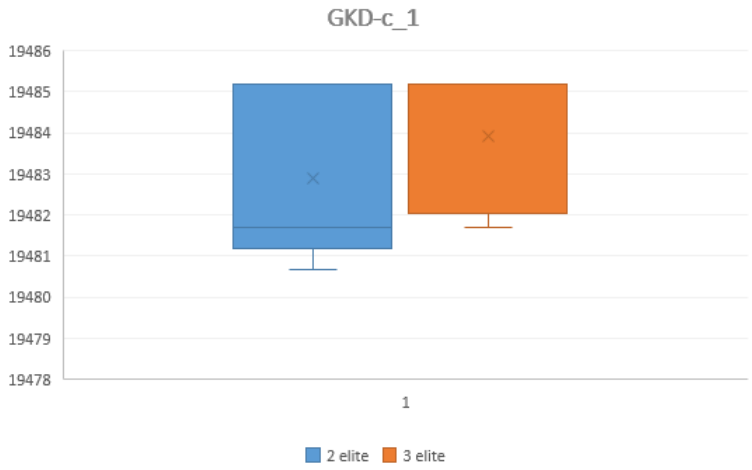


Figure 12: Costes obtenidos para el archivo GKD-c1, con una élite de 2 y 3

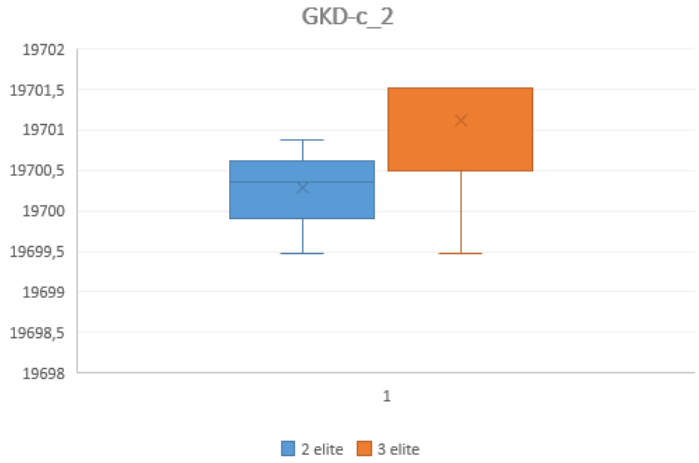


Figure 13: Costes obtenidos para el archivo GKD-c2, con una élite de 2 y 3

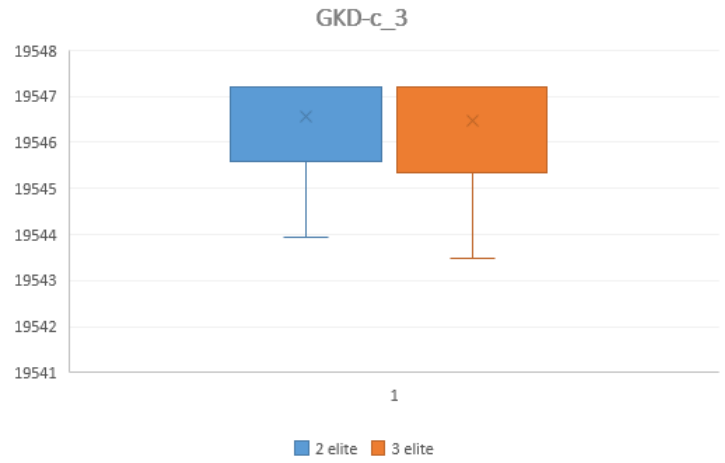


Figure 14: Costes obtenidos para el archivo GKD-c3, con una élite de 2 y 3

El operador de cruce MPX aplicado a la serie de datos SOM, con una élite de 2 o 3 individuos, ofrece unos resultados muy similares respecto a coste.

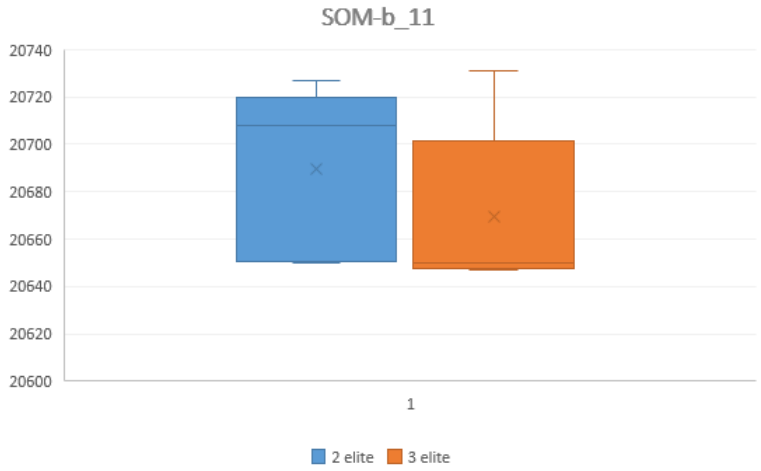


Figure 15: Costes obtenidos para el archivo SOM-b11, con una élite de 2 y 3

Figure 16: Costes obtenidos para el archivo SOM-b12, con una élite de 2 y 3

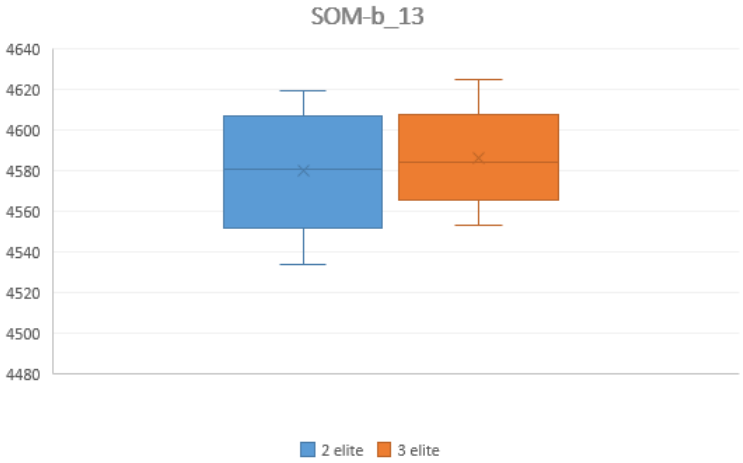


Figure 17: Costes obtenidos para el archivo SOM-b13, con una élite de 2 y 3

4.3.3 Cruce en dos puntos con elitismo 2 vs elitismo 3

4.3.4 Cruce Mpx con elitismo 3 vs dos puntos con elitismo 3

Si evaluamos los costes obtenidos, podemos observar que los dos operadores de cruce se comportan de manera similar en las instancias de datos GKD, no obstante, para el resto de datos, el cruce MPX ofrece unos resultados considerablemente mejores respecto al cruce en dos puntos, esto puede deberse a la

forma de reparación de los cromosomas, la cuál, difiere entre un tipo de cruce y otro.

El cruce MPX genera más genes de los deseados, y para reparar los cromosomas, se emplea una estrategia de reparación basada en la eliminación de los genes que menos aportan al coste de la solución. Sin embargo, el cruce en dos puntos, genera menos genes de los deseados, y la estrategia empleada es la de incluir los genes que más aportan que no están contenidos en la solución.

Si procedemos a evaluar la cantidad de tiempo empleada en obtener las soluciones, podemos observar que el cruce en dos puntos es hasta un 60% más lento respecto al cruce mpx. La diferencia de tiempos, viene determinada por la estrategia de reparación de los cromosomas.

Como se ha indicado anteriormente, dicha estrategia difiere según el tipo de cruce. Para el cruce en dos puntos se necesita recorrer todos los elementos no seleccionados, e ir evaluando el coste que aportan, este proceso se repite hasta que los cromosomas tengan los genes necesarios, sin embargo, en el mpx solo necesitamos eliminar elementos del cromosoma que menos aportan. Por tanto, el número de operaciones necesarias es mayor en el caso del cruce en dos puntos.

References

[1] Umbarkar, Dr. Anantkumar & Sheth, P.. (2015). CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A REVIEW. ICTACT Journal on Soft Computing (Volume: 6 , Issue: 1). 6. 10.21917/ijsc.2015.0150.

[2] https://sci2s.ugr.es/graduateCourses/Metaheuristicas

[3] https://sci2s.ugr.es/graduateCourses/Algoritmica