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

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(poblacion)
while evaluacionesRelizadas<evaluacionesLimite do
  poblacionPadres ← SeleccionaPoblacion(poblacion, semilla)
  poblacionHijos ← CruzarPoblacion(poblacionPadres, semilla)
  Reparar(poblacionHijos)
  MutarPoblacion(poblacionHijos,semilla)
  Evaluacion(poblacionHijos)
  poblacion ← 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 "poblacion". Esta tarea sera encomendada a la función "Evaluacion(poblacion)".

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(poblacion,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 "poblacion". 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 cinco 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 al reducir o incluso eliminar el componente de explotación de los algoritmos genéticos, 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 tres 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 la probabilidad de mutación.

Hemos utilizado los resultados obtenidos para realizar los siguientes gráficos de cajas y bigotes (Figuras 3, 4 y 5) y de barras (Figura 2).

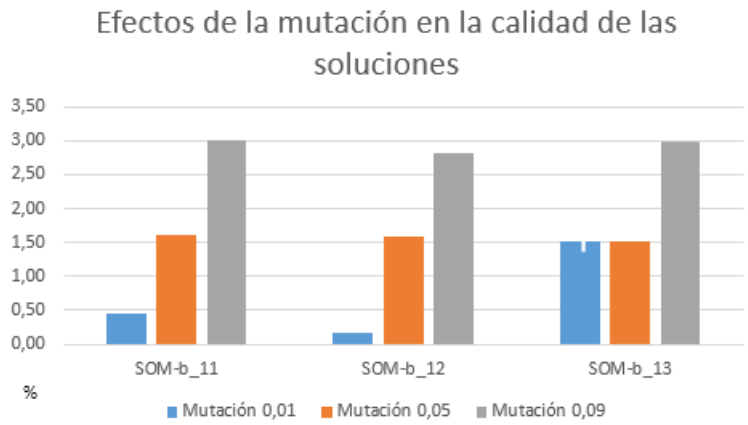

Figure 2: Diferencias en porcentajes respecto a los óptimos globales

En la Figura 2 tenemos una gráfica de barras que compara el porcentaje que difieren los resultados obtenido para las tres asignaciones del valor de probabilidad de mutación.

Como se puede observar en el gráfico anterior, a mayor porcentaje de mutación, mayor empeoramiento de los resultados obtenidos respecto al óptimo global en la ejecución de algoritmos genéticos.

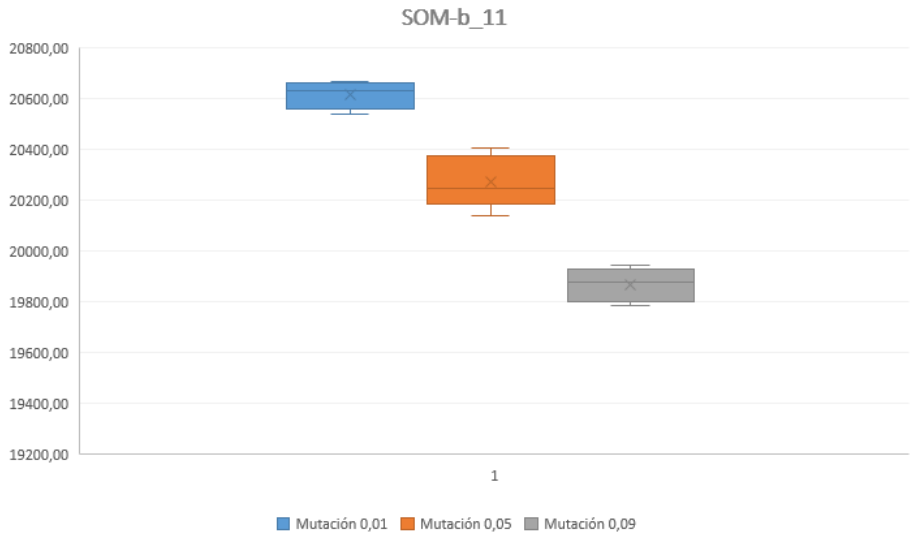

Figure 3: Gráfico de bigotes del problema SOM-b11

Figure 4: Gráfico de bigotes del problema SOM-b12

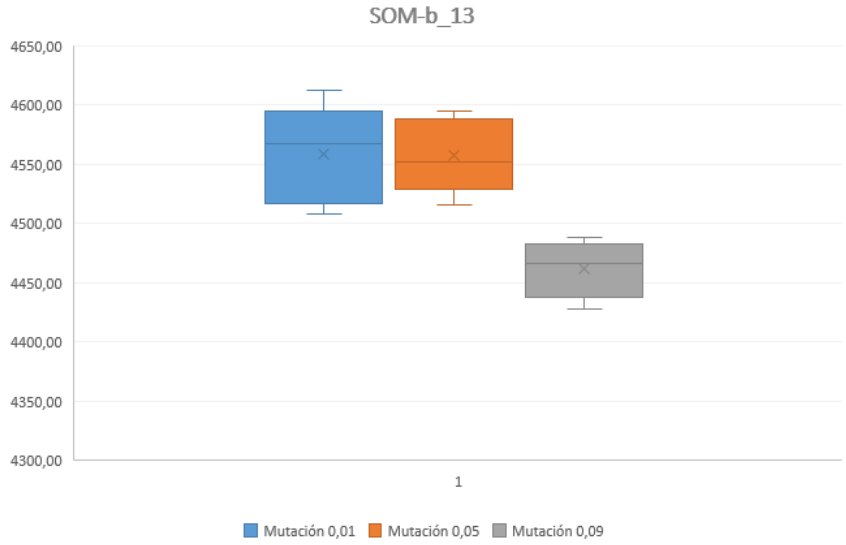

Figure 5: Gráfico de bigotes del problema SOM-b13

En las Figuras 3, 4 y 5 podemos observar la dispersión de los costes obtenidos. En estos se observa cómo el valor de mutación obtiene mejores resultados en todos los problemas es 0,01, y cómo estos difieren muy poco en el caso de los problemas SOM-b-11 y SOM-b-12, obteniendo no obstante una variación notable en el caso del problema SOM-b-13. Le sigue el valor de mutación 0,05 cuyo agrupamiento de los costes obtenidos empeora respecto al primer valor. El valor 0,09 es el que peor soluciones obtiene en todos los problemas y, además, es el valor cuyo gráfico es más compacto.

A raíz de los resultados obtenidos anteriormente, se reflexionó sobre el impacto que tendría fijar una capacidad de mutación nula. Al realizar las mismas 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, siendo incapaz de salir del estancamiento en óptimos locales.

A continuación, presentamos los resultados de las pruebas haciendo uso de gráficas de cajas y bigotes (Figura 6, 7 y 8). En ellos podemos observar que en los problemas SOM-b-11 y SOM-b-12 empeora únicamente los resultados obtenido con el valor de mutación a un 0,01. En el problema SOM-b-13 empeora a todos los valores de mutación, excepto al valor 0,09. Podemos observar también que en los problemas SOM-b-11 y SOM-b-12 se obtienen valores atípicos.

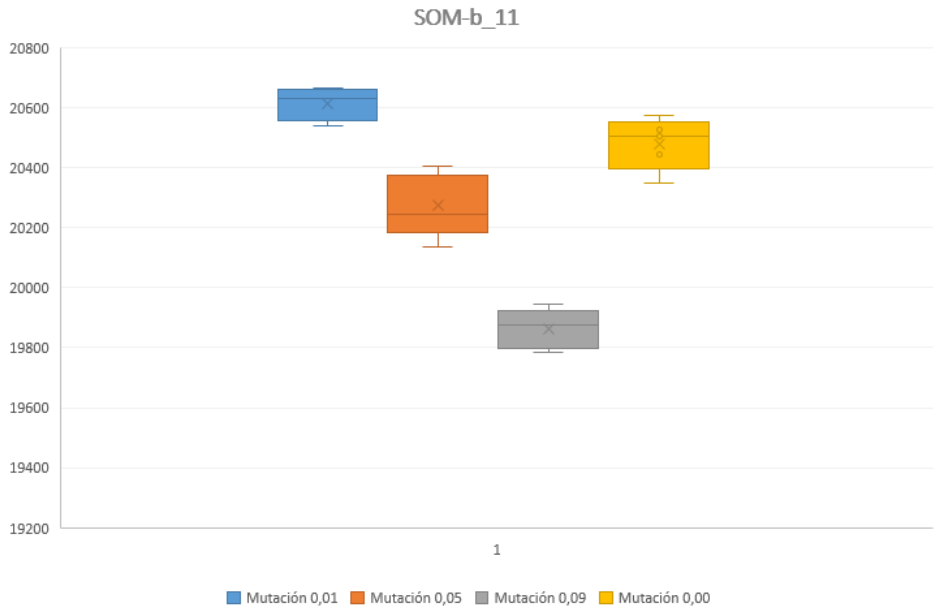

Figure 6: Gráfico de bigotes del problema SOM-b11

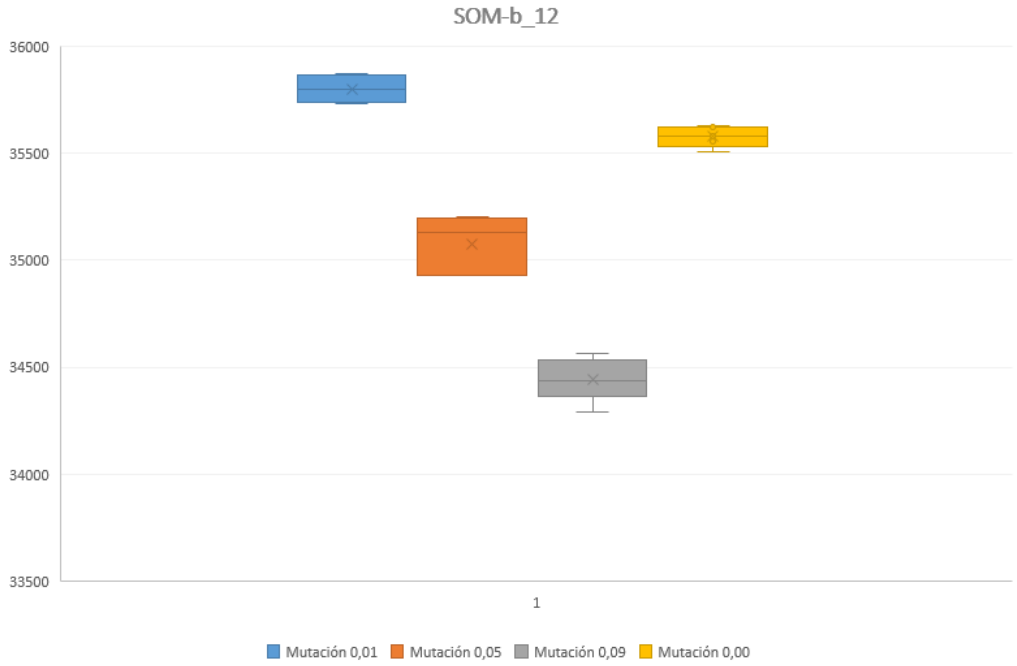

Figure 7: Gráfico de bigotes del problema SOM-b12

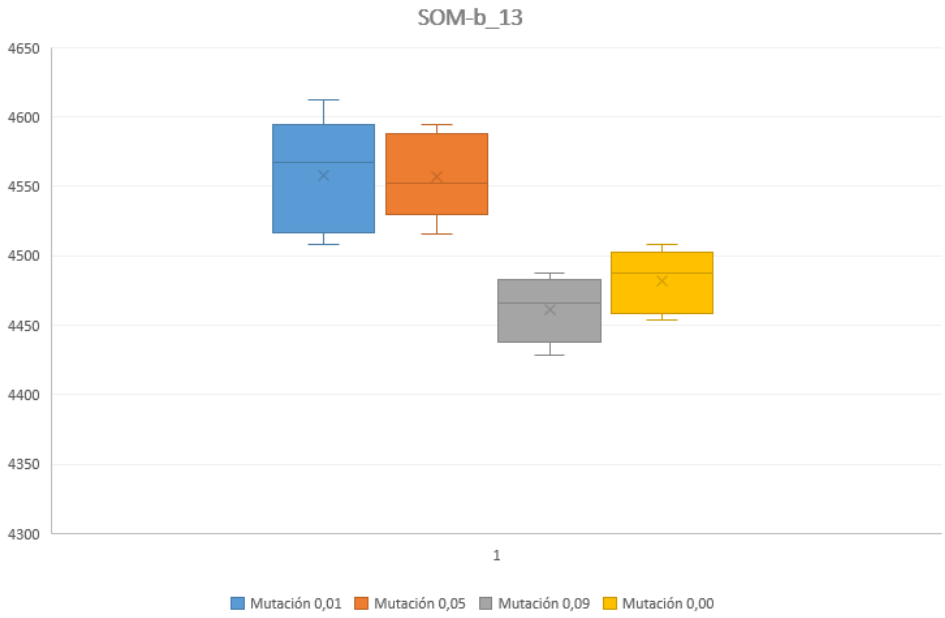

Figure 8: Gráfico de bigotes del problema SOM-b13

Basándonos en las pruebas realizadas y los resultados obtenidos de estas, llegamos a la determinación de que el mejor valor para la probabilidad de mutación del algoritmo genético para realizar la experimentación es 0,01.

4.3.2 Resultados obtenidos durante la experimentación

En las 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 cada uno de los problemas. La semilla utilizada ha sido 35608477.

Hemos decidido que la unidad de tiempo sea el número de evaluaciones realizadas ya que este valor es sobre el que se comprueba la condición de parada, además, puede arrojar información relevante a la hora de la modificación de dicho parámetro para obtener mejoras de tiempos o en el valor de las soluciones obtenidas.

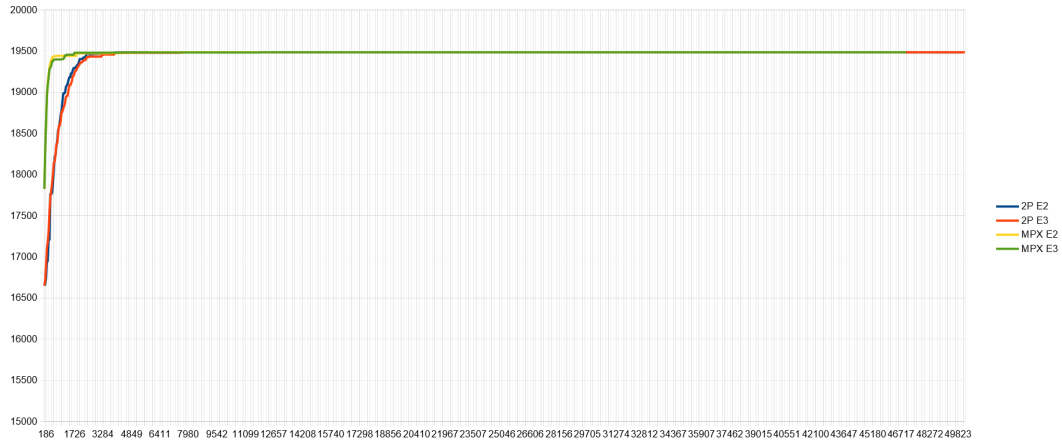

Figure 9: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema GKD-c1, semilla 35608477

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.

En la ejecución del algoritmo MPX con un elitismo = 2, el mejor coste obtenido se estabiliza en 19.485,19 a partir de las 6.250 evaluaciones, permaneciendo inalterable hasta finalizar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 400 evaluaciones.

En la ejecución del algoritmo MPX con un elitismo = 3, el mejor coste obtenido se estabiliza en 19.485,19 a partir de las 12.700 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 450 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 2, el mejor coste obtenido se estabiliza en 19.485,19 a partir de las 4.419 evaluaciones, permaneciendo estable hasta finalizar las 50.000 evaluaciones objetivo.Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 1.679 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 3, el mejor coste obtenido se estabiliza en 19.485,19 a partir de las 8.452 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 1.835 evaluaciones.

En todos los algoritmos ejecutados obtenemos un coste final = 19.485,19, coincidiendo con el óptimo global.

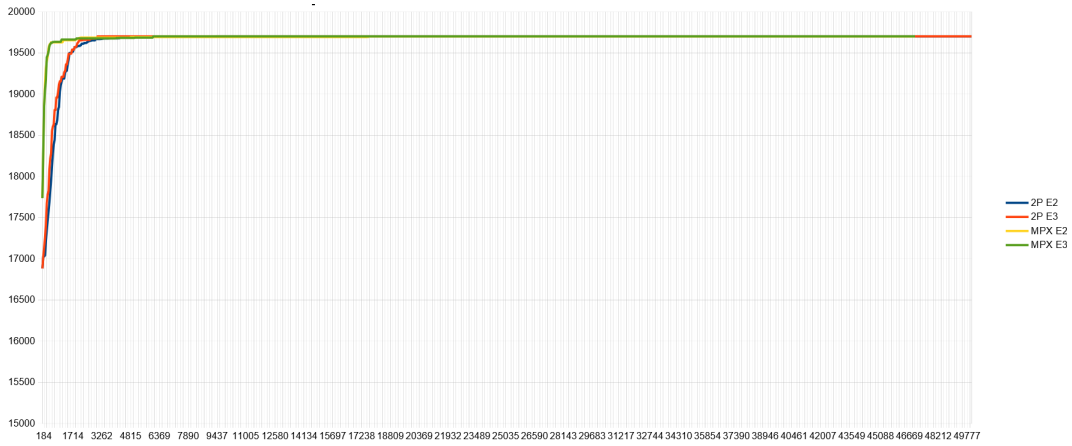

Figure 10: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema GKD-c2, semilla 35608477

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.

En la ejecución del algoritmo MPX con un elitismo = 2, el mejor coste obtenido se estabiliza en 19.700,34 a partir de las 18.800 evaluaciones, permaneciendo inalterable hasta finalizar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 450 evaluaciones.

En la ejecución del algoritmo MPX con un elitismo = 3, el mejor coste obtenido se estabiliza en 19.701,523 a partir de las 6.600 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 450 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 2, el mejor coste obtenido se estabiliza en 19.701,523 a partir de las 9.396 evaluaciones, permaneciendo estable hasta finalizar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 1.665 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 3, el mejor coste obtenido se estabiliza en 19.701,523 a partir de las 4.108 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 1.690 evaluaciones.

En todos los algoritmos ejecutados, menos en el MPX con elitismo = 2, obtenemos un coste final = 19.701,523 , difiriendo en un 0,0000008% aproximadamente respecto al óptimo global = 19.701,537.

En el caso del algoritmo MPX con elitismo = 3, obtenemos un coste final = 19.700,34. Este coste difiere un 0,00007% del óptimo global.

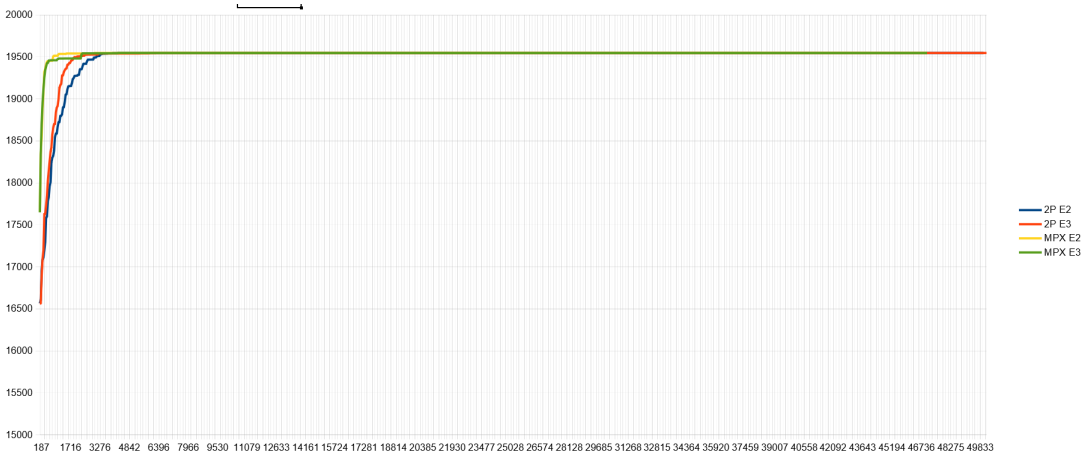

Figure 11: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema GKD-c3, semilla 35608477

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.

En la ejecución del algoritmo MPX con un elitismo = 2, el mejor coste obtenido se estabiliza en 19.547,215 a partir de las 3.200 evaluaciones, per-

maneciendo inalterable hasta finalizar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 400 evaluaciones.

En la ejecución del algoritmo MPX con un elitismo = 3, el mejor coste obtenido se estabiliza en 19.547,215 a partir de las 4.500 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 450 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 2, el mejor coste obtenido se estabiliza en 19.547,215 a partir de las 4.459 evaluaciones, permaneciendo estable hasta finalizar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 2.190 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 3, el mejor coste obtenido se estabiliza en 19.547,215 a partir de las 6.414 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 1.426 evaluaciones.

En todos los algoritmos ejecutados obtenemos un coste final = 19.547,215, coincidiendo con el óptimo global.

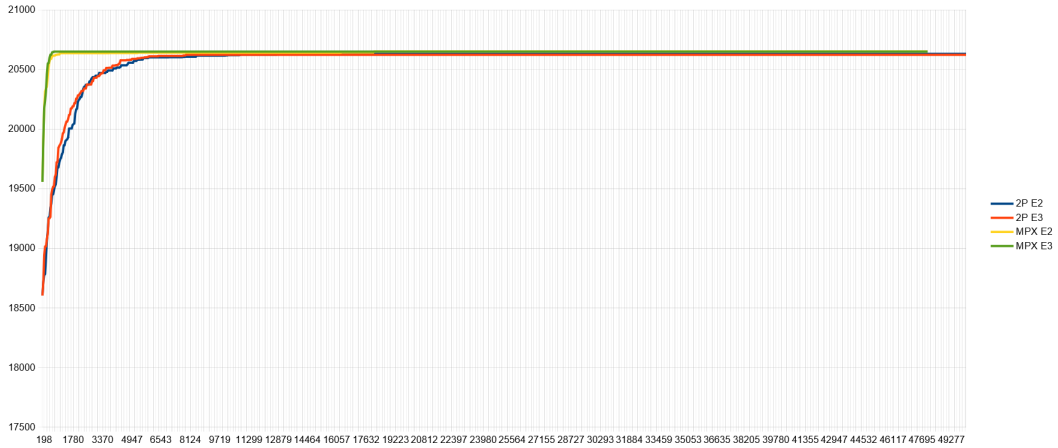

Figure 12: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema SOM-b11, semilla 35608477

Para el problema SOM-b11 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 148 evaluaciones para los dos elitismos. En este problema las soluciones óptimas son todas aquellas cuyo coste es superior a 18.668,7.

En la ejecución del algoritmo MPX con un elitismo = 2, el mejor coste obtenido se estabiliza en 20.651 a partir de las 18.800 evaluaciones, permaneciendo inalterable hasta finalizar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 500 evaluaciones.

En la ejecución del algoritmo MPX con un elitismo = 3, el mejor coste obtenido se estabiliza en 20.650 a partir de las 750 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 400 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 2, el mejor coste obtenido se estabiliza en 20.631 a partir de las 15.280 evaluaciones, permaneciendo estable hasta finalizar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 4.706 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 3, el mejor coste obtenido se estabiliza en 20.623 a partir de las 7.865 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 4.083 evaluaciones.

En el algoritmo MPX con un elitismo = 2 obtenemos un coste final = 20.651, que difiere del óptimo global en un 0,005% aproximadamente. El óptimo global para este problema es 20.743.

En el algoritmo MPX con un elitismo = 3 obtenemos un coste final = 20.650, que difiere del óptimo global en un 0,005% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 2 obtenemos un coste final = 20.631, que difiere del óptimo global en un 0,006% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 3 obtenemos un coste final = 20.623, que difiere del óptimo global en un 0,006% aproximadamente.

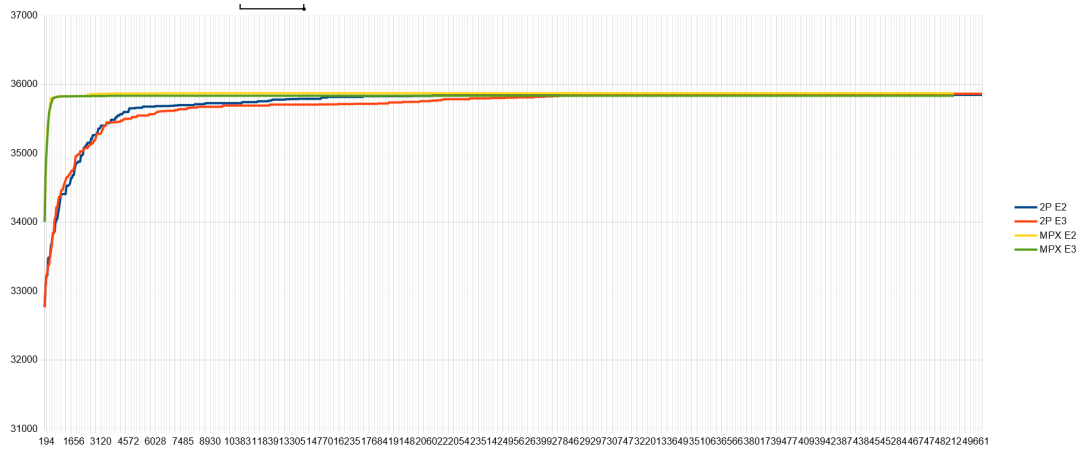

Figure 13: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema SOM-b12, semilla 35608477

Para el problema SOM-b12 todos los algoritmos empiezan a encontrar soluciones aceptables en torno a las 100 evaluaciones realizadas para los dos elitismos. En este problema las soluciones óptimas son todas aquellas cuyo coste es superior a 32.292,9.

En la ejecución del algoritmo MPX con un elitismo = 2, el mejor coste obtenido se estabiliza en 35.871 a partir de las 8.400 evaluaciones, permaneciendo inalterable hasta finalizar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 350 evaluaciones.

En la ejecución del algoritmo MPX con un elitismo = 3, el mejor coste obtenido se estabiliza en 35.834 a partir de las 3.550 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 350 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 2, el mejor coste obtenido se estabiliza en 35.847 a partir de las 21.669 evaluaciones, permaneciendo estable hasta finalizar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 3.993 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 3, el mejor coste obtenido se estabiliza en 35.863 a partir de las 39.724 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 4.746 evaluaciones.

En el algoritmo MPX con un elitismo = 2 obtenemos un coste final = 35.871, que difiere del óptimo global en un 0,0003% aproximadamente. El óptimo global para este problema es 35.881.

En el algoritmo MPX con un elitismo = 3 obtenemos un coste final = 35.834, que difiere del óptimo global en un 0,002% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 2 obtenemos un coste final = 35.847, que difiere del óptimo global en un 0,001% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 3 obtenemos un coste final = 35.863, que difiere del óptimo global en un 0,0006% aproximadamente.

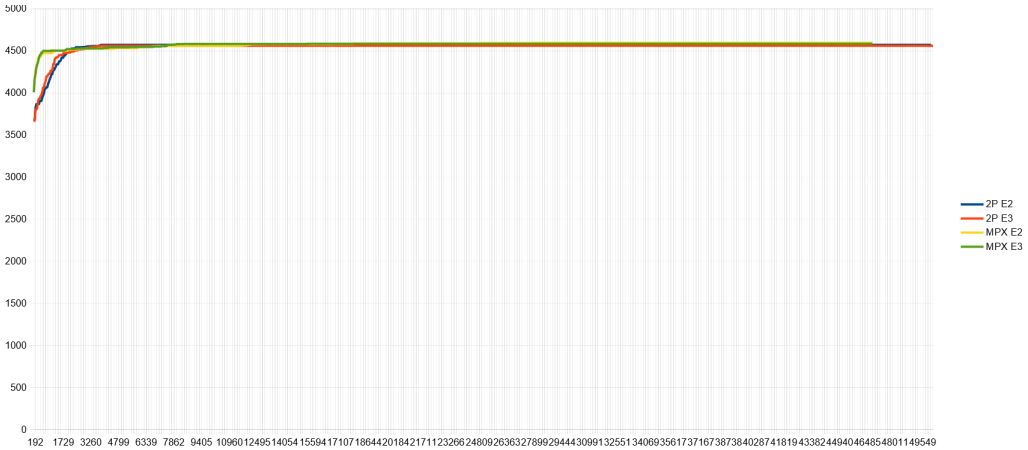

Figure 14: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema SOM-b13, semilla 35608477

Para el problema SOM-b13 los algoritmos MPX empiezan a encontrar soluciones aceptables en torno a las 200 evaluaciones realizadas para los dos elitismos. Los algoritmos de cruce en dos puntos empiezan a obtener soluciones aceptables a partir de las 1085 evaluaciones para una élite = 2, y a partir de las 793 evaluaciones para una élite = 3. En este problema las soluciones óptimas son todas aquellas cuyo coste es superior a 4.192,2.

En la ejecución del algoritmo MPX con un elitismo = 2, el mejor coste obtenido se estabiliza en 4.595 a partir de las 29.800 evaluaciones, permaneciendo inalterable hasta finalizar las 50.000 evaluaciones objetivo.

En la ejecución del algoritmo MPX con un elitismo = 3, el mejor coste obtenido se estabiliza en 4.584 a partir de las 16.400 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 2, el mejor coste obtenido se estabiliza en 4.571 a partir de las 3.818 evaluaciones, permaneciendo estable hasta finalizar las 50.000 evaluaciones objetivo.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 3, el mejor coste obtenido se estabiliza en 4.560 a partir de las 3.829 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo.

En este caso, ninguno de los algoritmos es capaz de encontrar costes que difieran menos de un 1% respecto al óptimo global.

En el algoritmo MPX con un elitismo = 2 obtenemos un coste final = 4.595, que difiere del óptimo global en un 0,02% aproximadamente. El óptimo global para este problema es 4.658.

En el algoritmo MPX con un elitismo = 3 obtenemos un coste final = 4.584, que difiere del óptimo global en un 0,02% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 2 obtenemos un coste final = 4.571, que difiere del óptimo global en un 0,02% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 3 obtenemos un coste final = 4.560, que difiere del óptimo global en un 0,03% aproximadamente.

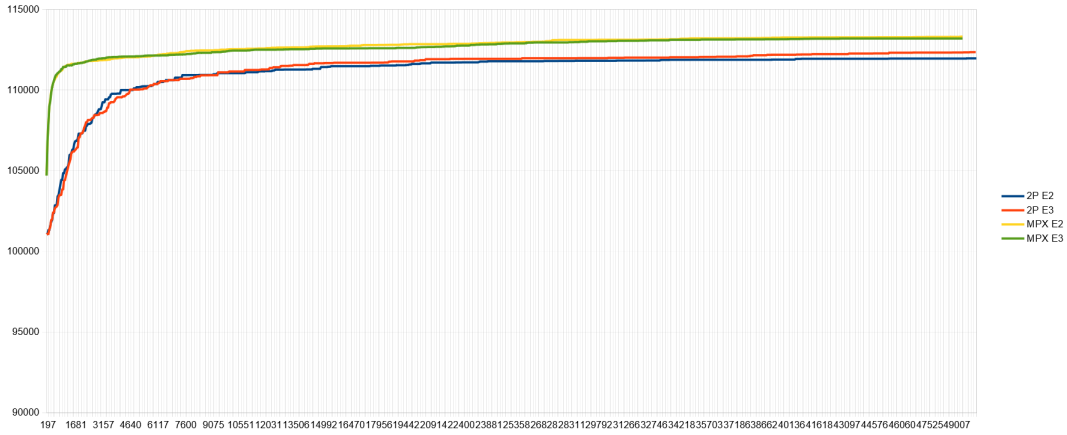

Figure 15: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema MDG-a21, semilla 35608477

Para el problema MDG-a21 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 545 evaluaciones para una élite = 2, y a partir de las 692 evaluaciones para una élite = 3. En este problema las soluciones óptimas son todas aquellas cuyo coste es superior a 102.833,1.

En la ejecución del algoritmo MPX con un elitismo = 2, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 113.312 al llegar a las 49.900 evaluaciones realizadas. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 31.700 evaluaciones.

En la ejecución del algoritmo MPX con un elitismo = 3, el mejor coste obtenido se estabiliza en 113.193 a partir de las 43.350 evaluaciones, permaneciendo inalterable hasta alcanzar las 50.000 evaluaciones objetivo. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 35.750 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 2, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 111.963 al llegar a las 49.554 evaluaciones realizadas.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 3, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 112.350 al llegar a las 49.926 evaluaciones realizadas.

En este caso, ningún algoritmo de corte en dos puntos consigue encontrar costes que difieran menos de un 1% respecto al óptimo global.

En el algoritmo MPX con un elitismo = 2 obtenemos un coste final = 113.312, que difiere del óptimo global en un 0,008% aproximadamente. El óptimo global para este problema es 114.259.

En el algoritmo MPX con un elitismo = 3 obtenemos un coste final = 113.193, que difiere del óptimo global en un 0,009% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 2 obtenemos un coste final = 111.963, que difiere del óptimo global en un 0,02% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 3 obtenemos un coste final = 112.350, que difiere del óptimo global en un 0,02% aproximadamente.

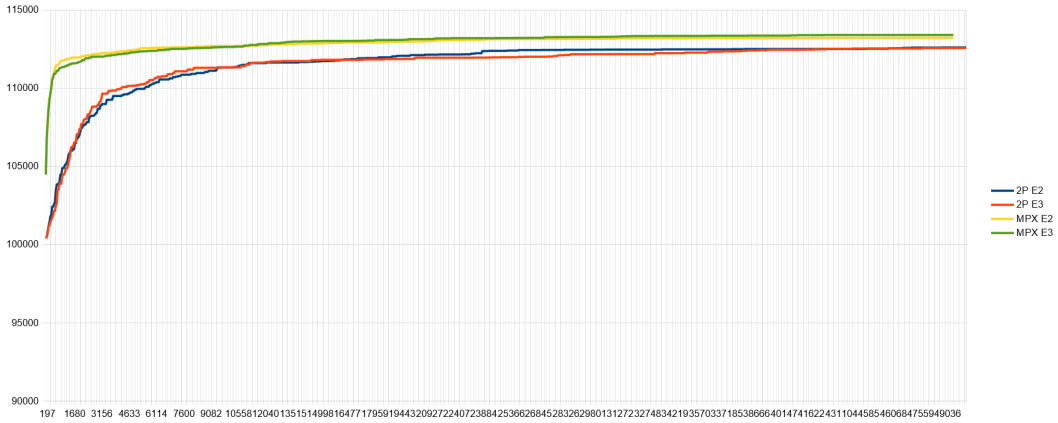

Figure 16: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema MDG-a22, semilla 35608477

Para el problema MDG-a22 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 642 evaluaciones para una élite = 2, y a partir de las 738 evaluaciones para una élite = 3. En este problema las soluciones óptimas son todas aquellas cuyo coste es superior a 102.894,3.

En la ejecución del algoritmo MPX con un elitismo = 2, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 113.217 al llegar a las 47.350 evaluaciones realizadas. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 31.550 evaluaciones.

En la ejecución del algoritmo MPX con un elitismo = 3, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 113.402 al llegar a las 42.950 evaluaciones realizadas.Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 21.850 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 2, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 112.604 al llegar a las 48.692 evaluaciones realizadas.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 3, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 112.564 al llegar a las 47.679 evaluaciones realizadas.

En este caso, ningún algoritmo de corte en dos puntos consigue encontrar costes que difieran menos de un 1% respecto al óptimo global.

En el algoritmo MPX con un elitismo = 2 obtenemos un coste final = 113.217, que difiere del óptimo global en un 0,01% aproximadamente. El óptimo global para este problema es 114.327.

En el algoritmo MPX con un elitismo = 3 obtenemos un coste final = 113.402, que difiere del óptimo global en un 0,009% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 2 obtenemos un coste final = 112.604, que difiere del óptimo global en un 0,02% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 3 obtenemos un coste final = 112.564, que difiere del óptimo global en un 0,02% aproximadamente.

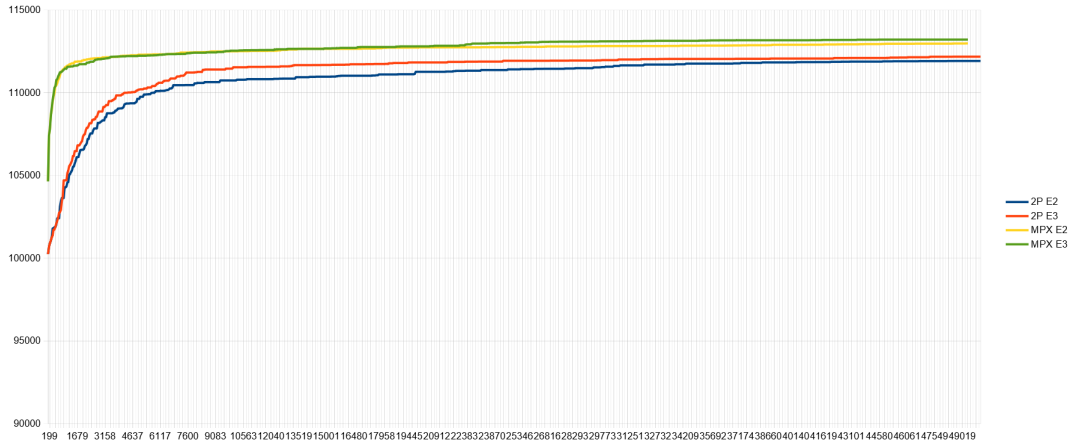

Figure 17: Evolución del mejor coste en la ejecución de todos los algoritmos respecto al número de evaluaciones para el problema MDG-a23, semilla 35608477

Para el problema MDG-a22 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 741 evaluaciones para una élite = 2, y a partir de las 692 evaluaciones para una élite = 3. En este problema las soluciones óptimas son todas aquellas cuyo coste es superior a 102.710,7.

En la ejecución del algoritmo MPX con un elitismo = 2, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 112.982 al llegar a las 49.900 evaluaciones realizadas. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 49.900 evaluaciones.

En la ejecución del algoritmo MPX con un elitismo = 3, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 113.212 al llegar a las 45.150 evaluaciones realizadas. Este algoritmo es capaz de encontrar costes con una diferencia inferior a un 1% respecto al óptimo global a partir de las 23.950 evaluaciones.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 2, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 111.921 al llegar a las 48.627 evaluaciones realizadas.

En la ejecución del algoritmo de corte en dos puntos con un elitismo = 3, no se estabiliza el mejor coste en un valor fijo, sino que se va mejorando continuamente, llegando a obtener el coste final obtenido = 112.178 al llegar a las 48.561 evaluaciones realizadas.

En este caso, ningún algoritmo de corte en dos puntos consigue encontrar costes que difieran menos de un 1% respecto al óptimo global.

En el algoritmo MPX con un elitismo = 2 obtenemos un coste final = 112.982, que difiere del óptimo global en un 0,01% aproximadamente. El óptimo global para este problema es 114.123.

En el algoritmo MPX con un elitismo = 3 obtenemos un coste final = 113.212, que difiere del óptimo global en un 0,008% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 2 obtenemos un coste final = 111.921, que difiere del óptimo global en un 0,02% aproximadamente.

En el algoritmo de corte en dos puntos con un elitismo = 3 obtenemos un coste final = 112.178, que difiere del óptimo global en un 0,02% aproximadamente.

4.3.3 Cruce MPX con elitismo 2 vs elitismo 3

De la experimentación realizada, obtenemos las siguientes tablas, correspondientes al cruce MPX:

Genético MPX elite 2						
	GKD-c_1_n500_m50		GKD-c_2_n500_m50		GKD-c_3_n500_m50	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	19481,688	892,00	19700,36	892,00	19547,215	888,00
2	19481,69	909,00	19700,36	897,00	19547,22	893,00
3	19485,19	918,00	19700,34	886,00	19547,22	894,00
4	19480,68	895,00	19699,48	1731,00	19543,93	895,00
5	19485,19	901,00	19700,88	905,00	19547,22	890,00
Media	-0,01%	903	-0,01%	1062	0,00%	892,00
Desviación Típica	0,01%	10,61	0,00%	373,94	0,01%	2,92

Table 1: Resultados GKD

Genético MPX elite 2						
	SOM-b_11_n300_m90		SOM-b_12_n300_m120		SOM-b_13_n400_m40	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	20708,00	2288,00	35866,00	4717,00	4619,00	597,00
2	20727,00	2404,00	35847,00	4535,00	4581,00	616,00
3	20651,00	2407,00	35871,00	4746,00	4595,00	604,00
4	20650,00	2260,00	35872,00	4546,00	4534,00	599,00
5	20713,00	2299,00	35844,00	4800,00	4570,00	606,00
Media	-0,26%	2331,60	-0,06%	4668,80	-1,68%	604,40
Desviación Típica	0,18%	68,95	0,04%	120,91	0,68%	7,44

Table 2: Resultados SOM

Genético MPX elite 2						
	MDG-a_21_n2000_m200		MDG-a_22_n2000_m200		MDG-a_23_n2000_m200	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	113460,00	30888,00	112852,00	29847,00	112933,00	29246,00
2	113135,00	28109,00	112957,00	28412,00	113487,00	28296,00
3	113312,00	28749,00	113217,00	28307,00	112982,00	29794,00
4	113112,00	27848,00	113059,00	29327,00	113251,00	29338,00
5	113088,00	28269,00	112829,00	28468,00	112789,00	30736,00
Media	-0,91%	28772,60	-1,18%	28872,20	-0,91	29482,00
Desviación Típica	0,14%	1227,23	0,14%	680,38	0,24%	887,82

Table 3: Resultados MDG

Genético MPX elite 3						
	GKD-c_1_n500_m50		GKD-c_2_n500_m50		GKD-c_3_n500_m50	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	19481,69	874,00	19701,52	884,00	19547,22	882,00
2	19482,37	872,00	19701,52	888,00	19547,22	1354,00
3	19485,19	892,00	19701,52	877,00	19547,22	892,00
4	19485,19	879,00	19699,48	883,00	19547,22	890,00
5	19485,19	875,00	19701,52	873,00	19543,48	881,00
Media	0,00%	1173,40	0,00%	1118,60	0,00%	979,80
Desviación Típica	0,01%	8,02	0,00%	5,96	0,01%	209,24

Table 4: Resultados GKD

Genético MPX elite 3						
	SOM-b_11_n300_m90		SOM-b_12_n300_m120		SOM-b_13_n400_m40	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	20648,00	2252,00	35870,00	4404,00	4590,00	1219,00
2	20731,00	2253,00	35866,00	4190,00	4625,00	612,00
3	20650,00	2128,00	35834,00	4424,00	4584,00	617,00
4	20672,00	2238,00	35879,00	4272,00	4553,00	602,00
5	20647,00	2215,00	35825,00	5704,00	4578,00	602,00
Media	-0,35%	2217,20	-0,07%	4598,80	-1,55%	730,40
Desviación Típica	0,17%	52,17	0,07%	625,28	0,56%	273,21

Table 5: Resultados SOM

Genético MPX elite 3						
	MDG-a_21_n2000_m200		MDG-a_22_n2000_m200		MDG-a_23_n2000_m200	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	112931,00	28983,00	113173,00	27009,00	112978,00	27013,00
2	113375,00	28527,00	113335,00	26306,00	113350,00	26662,00
3	113193,00	27110,00	113402,00	27403,00	113212,00	26661,00
4	113074,00	28232,00	112677,00	24702,00	113274,00	25797,00
5	113445,00	26549,00	113470,00	27643,00	112600,00	26081,00
Media	-0,92%	27880,20	-0,98%	26612,60	-0,91%	26442,80
Desviación Típica	0,19%	1015,33	0,28%	1181,95	0,27%	492,12

Table 6: Resultados MDG

El operador de cruce MPX aplicado a la serie de datos GKD, con una élite de 2 o 3 individuos, en términos de coste presenta unas diferencias ínfimas.

Para el archivo GKD-c1 (18), 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 (19),los resultados son ligeramente superiores con una élite de 3 individuos, no obstante, los resultados tienden a agruparse más con una élite de dos, pero sin llegara obtener el óptimo.

Para el archivo GKD-c3 (20), se comportan de manera muy similar.

En lo que se refiere a convergencia de los costes, el comportamiento de los dos operadores es prácticamente similar. Podemos observar este comportamiento en las Figuras 9, 10 y 11.

A continuación, mostramos gráficos de cajas y bigotes para cada problema de la serie GKD (Figura 18, 19 y 20). En ellos podemos comparar tanto los costes obtenidos como la agrupación de los mismos.

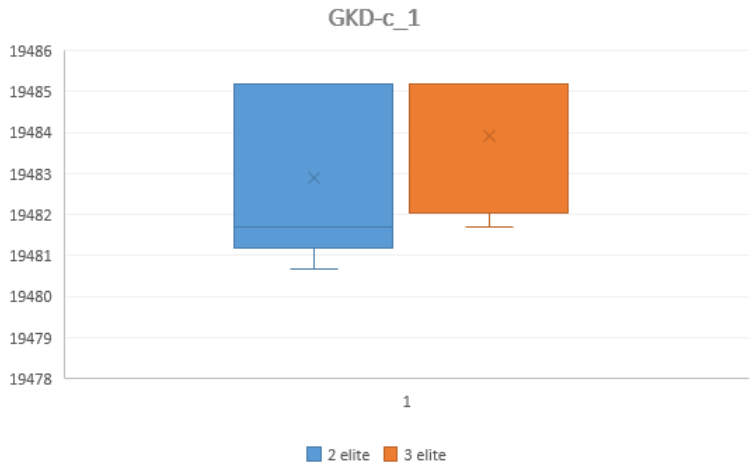

Figure 18: Costes obtenidos para el archivo GKD-c1, con una élite de 2 y 3

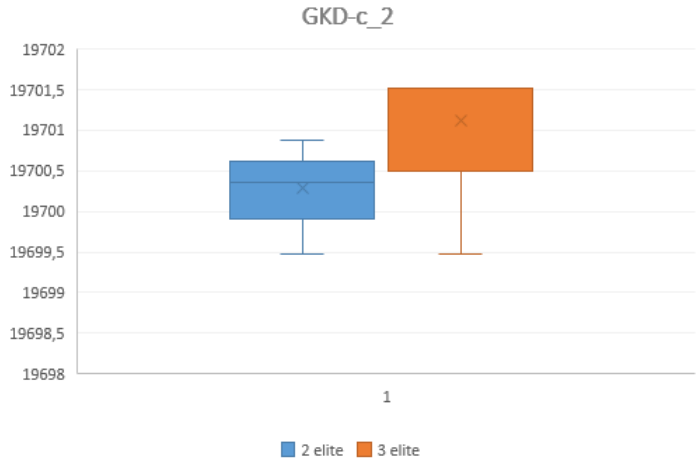

Figure 19: Costes obtenidos para el archivo GKD-c2, con una élite de 2 y 3

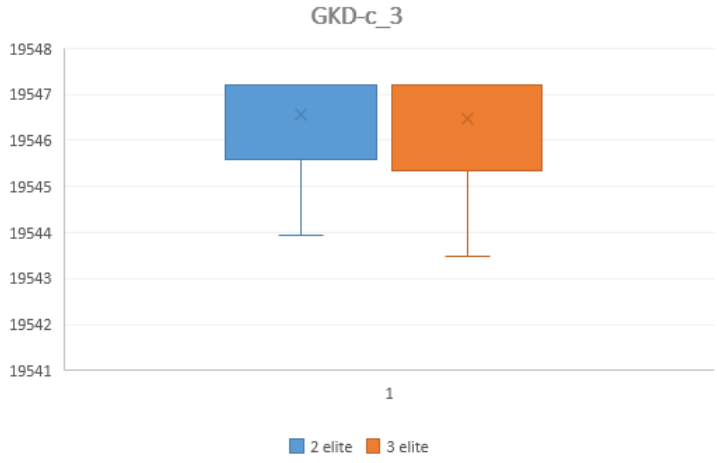

Figure 20: Costes obtenidos para el archivo GKD-c3, con una élite de 2 y 3

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.

En cuanto a la convergencia de los costes obtenidos, podemos observar en la Figura 12, 13 y 14 que el comportamiento de los algoritmos es prácticamente similar para los dos valores de elitismo.

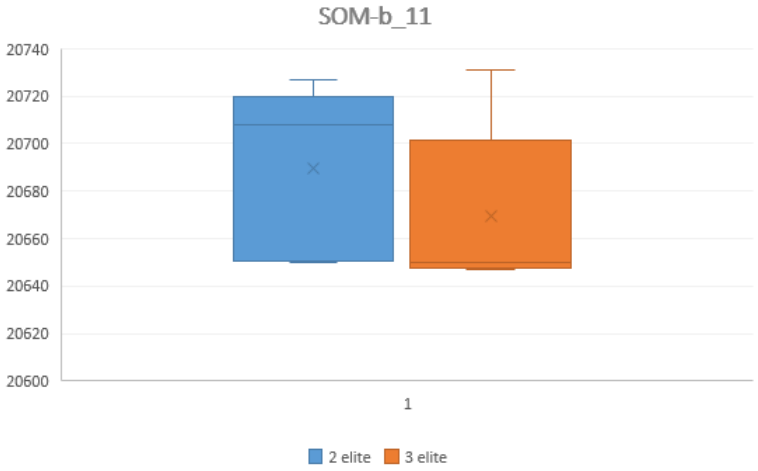

Figure 21: Costes obtenidos para el archivo SOM-b11, con una élite de 2 y 3

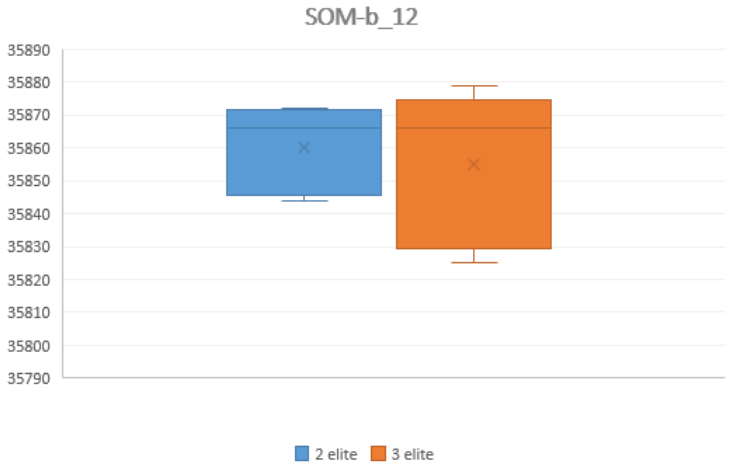

Figure 22: Costes obtenidos para el archivo SOM-b12, con una élite de 2 y 3

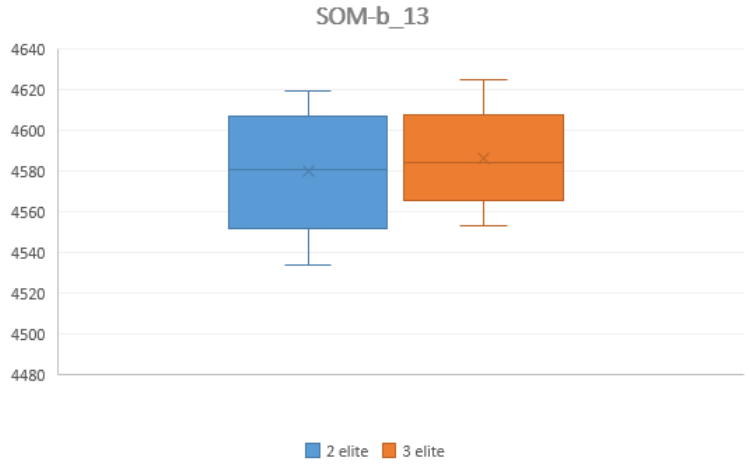

Figure 23: Costes obtenidos para el archivo SOM-b13, con una élite de 2 y 3

Como se ha podido observar en los resultados, no existe una diferencia significativa a nivel de costes, tiempo o convergencia de las soluciones, entre un elitismo de dos individuos y un elitismo de tres individuos. No obstante, teniendo en cuenta las ligeras diferencias observadas en la experimentación, elegimos al operador de cruce MPX con un valor de elitismo = 3 como ganador.

4.3.4 Cruce en dos puntos con elitismo 2 vs elitismo 3

De la experimentación realizada, obtenemos las siguientes tablas, correspondientes al cruce en dos puntos:

Genético 2 puntos elite 2						
	GKD-c_1_n500_m50		GKD-c_2_n500_m50		GKD-c_3_n500_m50	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	19485,19	1205,00	19700,36	1131,00	19547,22	1169,00
2	19485,19	1184,00	19701,52	1109,00	19547,22	1121,00
3	19485,19	1142,00	19701,52	1118,00	19547,22	1109,00
4	19485,19	1179,00	19701,52	1103,00	19547,22	1112,00
5	19485,19	1157,00	19700,36	1132,00	19547,22	1121,00
Media	0,00%	1173,40	0,00%	1118,60	0,00%	1126,40
Desviación Típica	0,00%	24,48	0,00%	12,93	0,00%	24,41

Table 7: Resultados GKD

Genético 2 puntos elite 2						
	SOM-b_11_n300_m90		SOM-b_12_n300_m120		SOM-b_13_n400_m40	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	20644,00	1807,00	35848,00	3839,00	4631,00	1211,00
2	20650,00	1752,00	35813,00	3751,00	4542,00	1054,00
3	20631,00	1786,00	35847,00	3449,00	4571,00	817,00
4	20674,00	1825,00	35803,00	3713,00	4614,00	646,00
5	20638,00	1800,00	35871,00	3847,00	4540,00	657,00
Media	-0,46%	1794,00	-0,12%	3719,80	-1,68%	877,00
Desviación Típica	0,08%	27,36	0,08%	161,82	0,89%	249,12

Table 8: Resultados SOM

Genético 2 puntos elite 2						
	MDG-a_21_n2000_m200		MDG-a_22_n2000_m200		MDG-a_23_n2000_m200	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	112242,00	44907,00	112506,00	46008,00	111991,00	44417,00
2	112280,00	43616,00	112443,00	43669,00	112497,00	43752,00
3	111963,00	45431,00	112604,00	44006,00	111921,00	44011,00
4	112012,00	42691,00	112723,00	43679,00	112302,00	43656,00
5	112367,00	44207,00	111886,00	43836,00	112163,00	43111,00
Media	-1,83%	44170,40	-1,66%	44239,60	-1,71%	43789,40
Desviación Típica	0,15%	1075,76	0,28%	998,07	0,20%	480,21

Table 9: Resultados MDG

Genético 2 puntos elite 3						
	GKD-c_1_n500_m50		GKD-c_2_n500_m50		GKD-c_3_n500_m50	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	19485,19	1081,00	19701,52	1048,00	19547,22	1049,00
2	19485,19	1080,00	19700,36	1037,00	19547,22	1079,00
3	19485,19	1054,00	19701,52	1012,00	19547,22	1069,00
4	19485,19	1085,00	19701,52	1030,00	19547,22	2404,00
5	19485,19	1061,00	19701,52	1038,00	19547,22	1079,00
Media	0,00%	1172,20	0,00%	1033,00	0,00%	1336,00
Desviación Típica	0,00%	13,77	0,00%	13,38	0,00%	597,16

Table 10: Resultados GKD

Genético 2 puntos elite 3						
	SOM-b_11_n300_m90		SOM-b_12_n300_m120		SOM-b_13_n400_m40	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	20636,00	1744,00	35801,00	2501,00	4588,00	632,00
2	20666,00	1660,00	35863,00	2520,00	4572,00	595,00
3	20623,00	1658,00	35863,00	3165,00	4560,00	587,00
4	20662,00	1720,00	35860,00	2531,00	4646,00	608,00
5	20689,00	1701,00	35863,00	2540,00	4573,00	588,00
Media	-0,42%	1696,60	-0,09%	2651,40	-1,51%	602,00
Desviación Típica	0,13%	37,56	0,08%	287,48	0,73%	18,75

Table 11: Resultados SOM

Genético 2 puntos elite 3						
	MDG-a_21_n2000_m200		MDG-a_22_n2000_m200		MDG-a_23_n2000_m200	
Ejecución	Coste	Tiempo	Coste	Tiempo	Coste	Tiempo
1	112586,00	44120,00	112506,00	44754,00	112402,00	41142,00
2	112538,00	43647,00	112210,00	44155,00	112253,00	40152,00
3	112350,00	42045,00	112564,00	43279,00	112178,00	40173,00
4	112488,00	41706,00	112444,00	43776,00	112437,00	40842,00
5	112420,00	42935,00	112352,00	42549,00	112500,00	40781,00
Media	-1,56%	42890.60%	-1,67%	43702,60	-1,55%	40618,00
Desviación Típica	0,08%	1025,17	0,12%	840,49	0,12%	437,75

Table 12: Resultados MDG

Como se puede ver en las anteriores tablas, el algoritmo genético con cruce en dos puntos, tiene un mejor comportamiento cuando se fija una élite de 3 individuos. Los resultados son mejores tanto en media, cómo en desviación típica, este último valor nos indica que con una élite de 3, el algoritmo es más robusto.

Las mayores diferencias las podemos encontrar en la serie de datos MDG, donde el cruce en dos puntos con una élite de 3 individuos, ofrece en la mayoría de casos, mejores resultados, y una agrupación de los resultados mayor. A continuación se incluyen los boxplots asociados a los archivos:

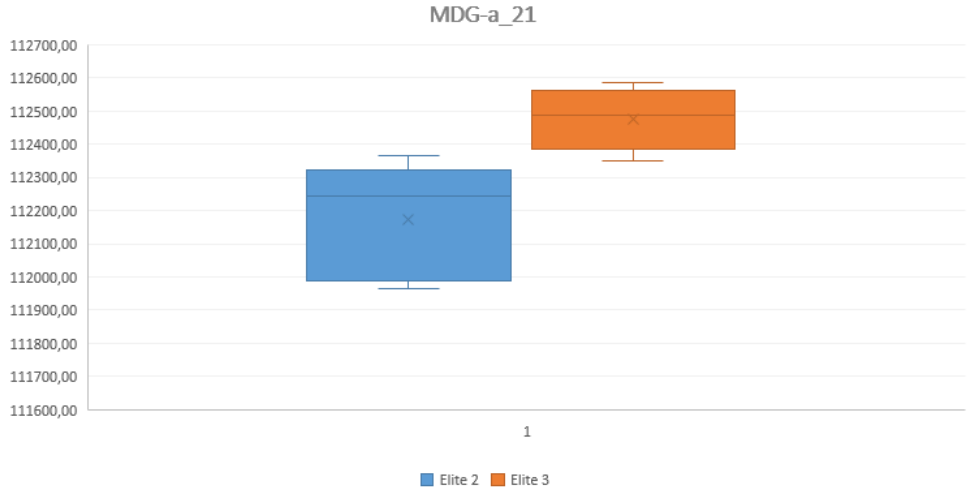

Figure 24

Como se puede observar en la gráfica 24, con una élite de 3 puntos se consiguen mejores resultados, y un mayor agrupamiento de los mismos.

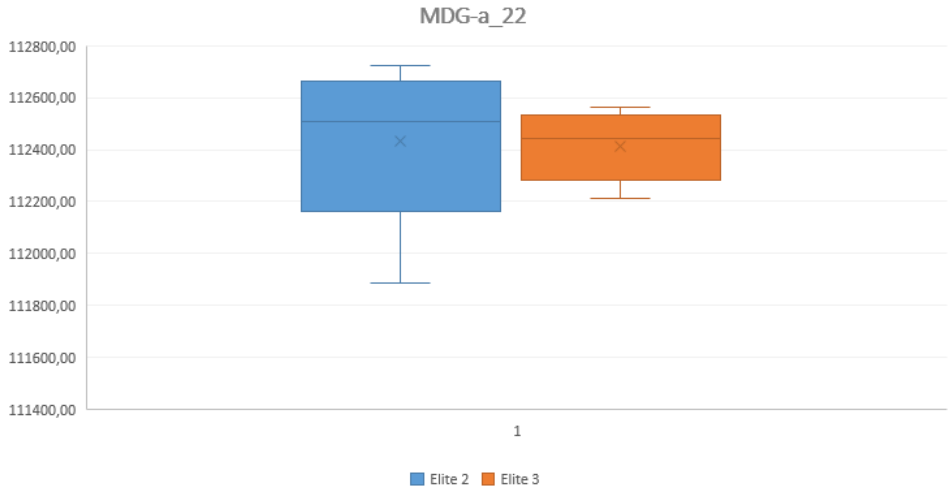

Figure 25

En el archivo MDG-a_2224, con una élite de 3 puntos se consigue un mayor agrupamiento de las soluciones, pero los máximos están por debajo de los encontrados con una élite de 2.

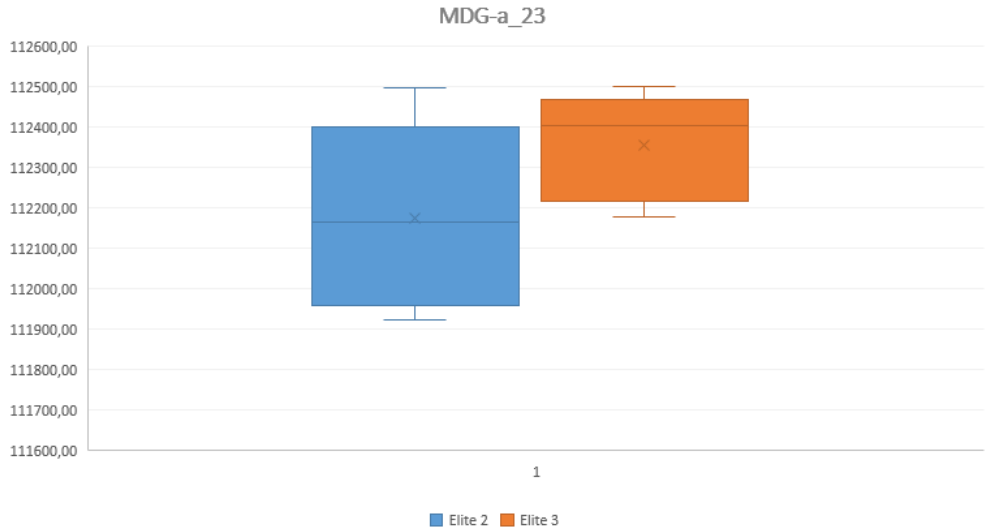

Figure 26

En el archivo MDG-a_2324, con una élite de 3 puntos se consigue un mayor agrupamiento de las soluciones.

Dada la ligera mejora que presenta el cruce en dos puntos con una élite de 3 individuos, lo elegimos como ganador de la comparativa.

4.3.5 Cruce Mpx con elitismo 3 vs cruce en dos puntos con elitismo 3

De las pruebas realizadas, se pueden extraer las siguientes tablas, donde para cada archivo y tipo de cruce, se indica la desviación típica de la media de las ejecuciones, con respecto a los óptimos globales, así como una media de los tiempos.

Tabla final						
	GKD-c_1_n500_m50		GKD-c_2_n500_m50		GKD-c_3_n500_m50	
MPX, 2 Elite	-0,01%	903,00	-0,01%	1062,20	0,00%	892,00
MPX, 3 Elite	-0,01%	878,40	0,00%	881,00	0,00%	979,80
2 puntos, 2 Elite	0,00%	1173,40	0,00%	1118,60	0,00%	1126,40
2 puntos, 3 Elite	0,00%	1072,20%	0,00%	1033,00	0,00%	1336,00

Table 13: Resultados GKD

Tabla final						
	SOM-b_11_n300_m90		SOM-b_12_n300_m120		SOM-b_13_n400_m40	
MPX, 2 Elite	-0,26%	2331,60	-0,06%	4668,80	-1,68	604,40
MPX, 3 Elite	-0,35%	2217,20	-0,07%	4598,80	-1,55%	730,40
2 puntos, 2 Elite	-0,46%	1794,00	-0,12%	3719,80	-1,68%	877,00
2 puntos, 3 Elite	-0,42%	1696,60%	-0,09%	2651,40	-1,51%	602,00

Table 14: Resultados SOM

Tabla final						
	MDG-a_21_n2000_m200		MDG-a_22_n2000_m200		MDG-a_23_n2000_m200	
MPX, 2 Elite	-0,91%	28772,60	-1,18%	28872,20	-0,91	29482,00
MPX, 3 Elite	-0,92%	27880,20	-0,98%	26612,60	-0,91%	26442,80
2 puntos, 2 Elite	-1,83%	44170,40	-1,66%	44239,60	-1,71%	43789,40
2 puntos, 3 Elite	-1,56%	42890.60%	-1,67%	43702,60	-1,55%	40618,00

Table 15: Resultados MDG

Tabla resumen		
	Coste	Tiempo
MPX, 2 Elite	-0,56%	10843,20
MPX, 3 Elite	-0,53%	10135,69
2 puntos, 2 Elite	-0,83%	15778,73
2 puntos, 3 Elite	-0,76%	15066,93

Table 16: Resultados

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 estrategia 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 a la solución, que no están contenidos en la solución.

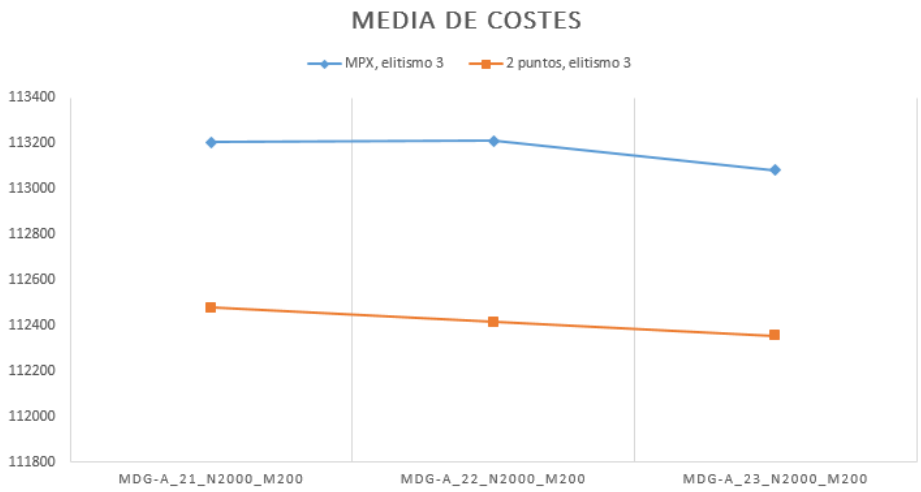

Figure 27: Media de tiempos para los distintos archivos MDG, tiempos expresados en milisegundos

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.

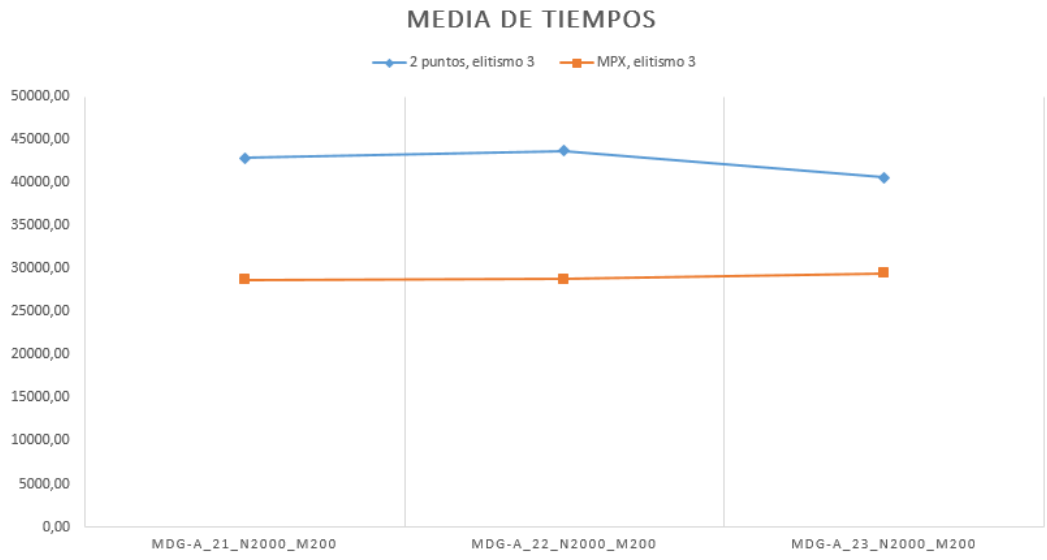

Figure 28: Media de tiempos para los distintos archivos MDG, tiempos expresados en milisegundos

Como se ha indicado anteriormente, dicha estrategia de reparación, 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 los genes del cromosoma que menos aportan. Por tanto, el número de operaciones necesarias es mayor en el caso del cruce en dos puntos.

4.3.6 Posibles mejoras

Cómo se ha visto anteriormente en las gráficas de convergencias asociadas a los archivos MDG, (15),(16) y (17), el cruce en dos puntos presenta una evolución más lenta en comparación con el cruce MPX. Para subsanar esa deficiencia, se han pensado una serie de cambios que serían interesantes aplicar:

- Aumentar el número máximo de evaluaciones: El algoritmo tendría la capacidad de mejorar, pero a costa de empeorar los tiempos.
- Uso de memorias adaptativas: Un sistema de memorias similar al de la búsqueda tabú podría ayudar a mejorar los resultados.
- Reiniciar parte de la población: Pasadas cierto número de generaciones sin mejora, se podría optar por reiniciar a una parte de los individuos de la población, para aumentar la diversidad.
- Cambiar la estrategia de reparación: Un Greedy puro, como se vio anteriormente, no ofrece los mejores resultados, podría sustituirse por un Greedy Randomized Adaptive Search Procedure (GRASP).

4.3.7 Conclusiones finales

Como se ha podido observar, el cruce MPX ofrece mejores resultados tanto en lo referente a costes de las soluciones finales, como en tiempos. A su vez, se ha podido comprobar mediante las gráficas de convergencias 17, que desde el primer momento, los resultados presentan una evolución más rápida. Por todos estos motivos, el cruce MPX con una élite de 2 individuos, es el mejor algoritmo que podemos aplicar a esta serie de datos.

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] A.J. Umbarkar, M.S. Joshi, Wei-Chiang Hong, Multithreaded Parallel Dual Population Genetic Algorithm (MPDPGA) for unconstrained function optimizations on multi-core system, Applied Mathematics and Computation, Volume 243, 2014, Pages 936-949, ISSN 0096-3003.