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

Algorithm 1 Algoritmo Genético

```
poblacion ← GeneraPoblacionInicial(semilla)
Evaluacion(poblacionPadres)
while evaluacionesRelizadas<evaluacionesLimite do
  poblacionPadres ← SeleccionaPoblacion(poblacion, 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)"

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.

Algorithm 3 Evaluacion(poblacion,obtenerElite)

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

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
```

Algorithm 5 CruzarPoblacion(poblacion,semilla)

```
poblacionHijos ← SeleccionaPoblacion(poblacion, semilla)
if tipoCruceMPX then
  poblacionHijos ← RealizaCruceMPX(poblacionHijos)
else
  poblacionHijos ← RelizaCruce2p(poblacionHijos)
end if
return poblacionHijos
```

Algorithm 6 SeleccionaPoblacion(poblacion,semilla)

```
while tamañoPoblacionHijos<numHijos do
  individuoSeleccionado ← SeleccionaIndividuo(poblacion, semilla)
  poblacionHijos ← poblacionHijos ∪ {individuoSeleccionado}
end while
return poblacionHijos
```

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
```

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
```

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
```

Algorithm 10 Reparar(poblacionHijos)

```
for individuo ∈ poblacionHijos do
  if !FuncionSolucion(individuo) then
    if tamañoIndividuo>tamañoIndividuoProblema then
      while tamañoIndividuo>tamañoIndividuoProblema do
        elementoMenor ← CalcularAportes(individuo)
        individuo ← individuo{elementoMenor}
      end while
    else if tamañoIndividuo<tamañoIndividuoProblema then
      elementoMayor ← CalcularMayorAporte(individuo)
      individuo ← individuo ∪ {elementoMayor}
    end if
  end if
end for
```

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)
```

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]
```

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]
```

Algorithm 14 Mutar(poblacionHijos,semilla)

```
for individuo ∈ poblacionHijos do
  posMuta ← GeneraEnteroAleatorio(semilla)
  eleMutado ← GeneraEnteroAleatorio(semilla)
  Intercambia(individuo,posMuta,eleMutado)
end for
```

Algorithm 15 ReemplazarElite(poblacionHijos,elites)

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

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.

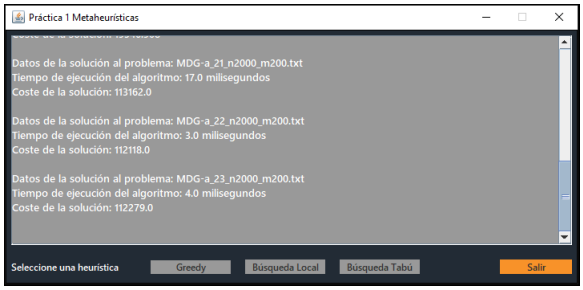


Figure 1: GUI

4.2 Parámetros de los algoritmos

4.2.1 Genetico

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 Operador de 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 2, 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 3,los resultados son ligeramente superiores con una élite de 3 individuos.

Para el archivo GKD-c3 4, los resultados revelan un comportamiento similar.

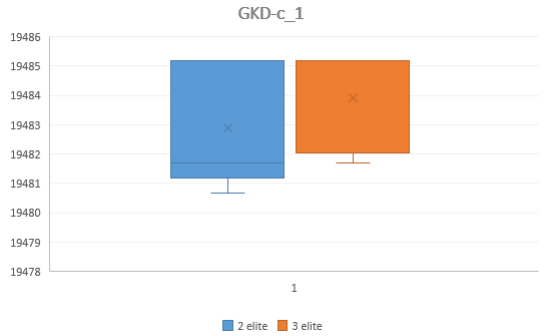


Figure 2: Costes obtenidos para el archivo GKD-c1, con una élite de 2 y 3

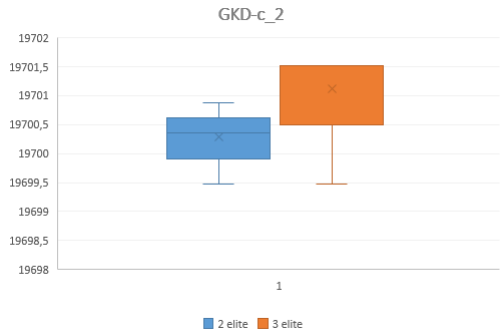


Figure 3: Costes obtenidos para el archivo GKD-c2, con una élite de 2 y 3

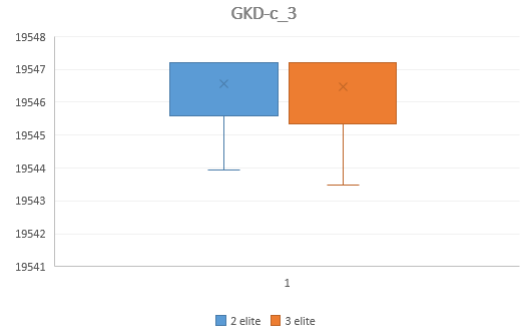


Figure 4: Costes obtenidos para el archivo GKD-c3, con una élite de 2 y 3

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.

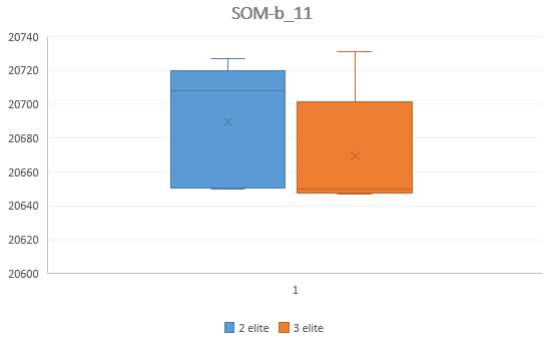


Figure 5: Costes obtenidos para el archivo SOM-b11, con una élite de 2 y 3

Figure 6: Costes obtenidos para el archivo SOM-b12, con una élite de 2 y 3

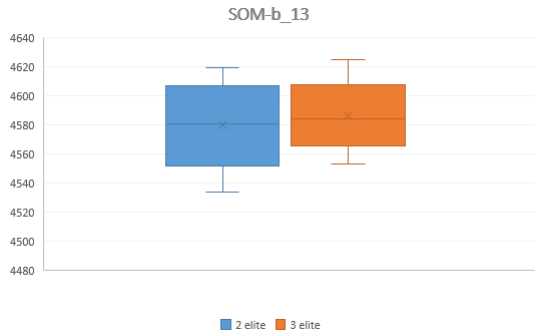


Figure 7: Costes obtenidos para el archivo SOM-b13, con una élite de 2 y 3

4.3.2 Efectos de la mutación

La mutación en los algoritmos genéticos es una técnica de diversificación, 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:

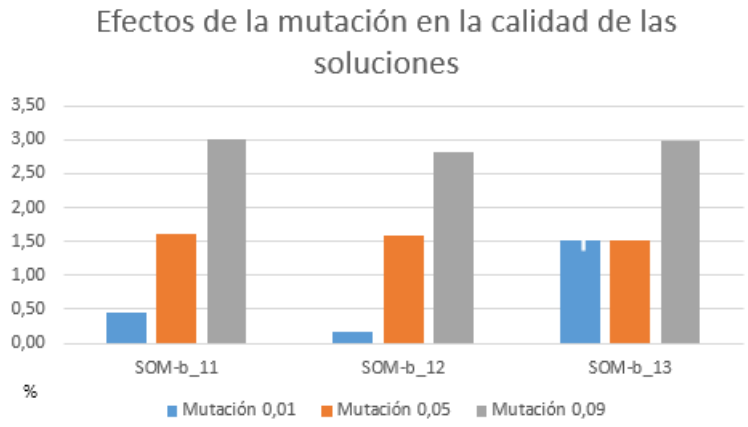
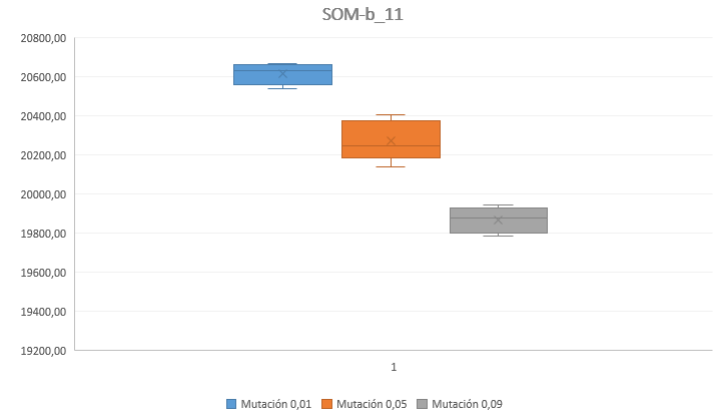
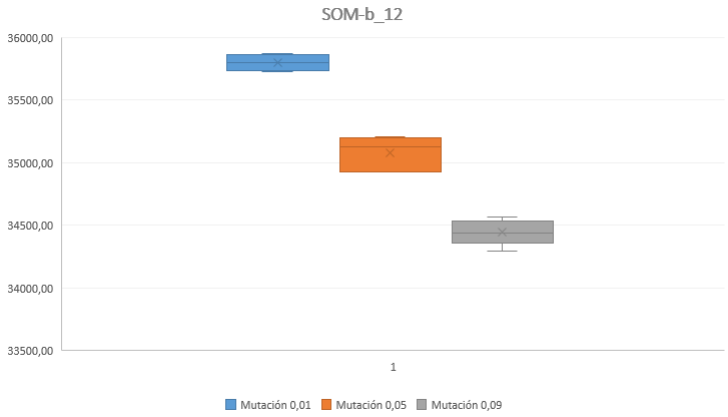
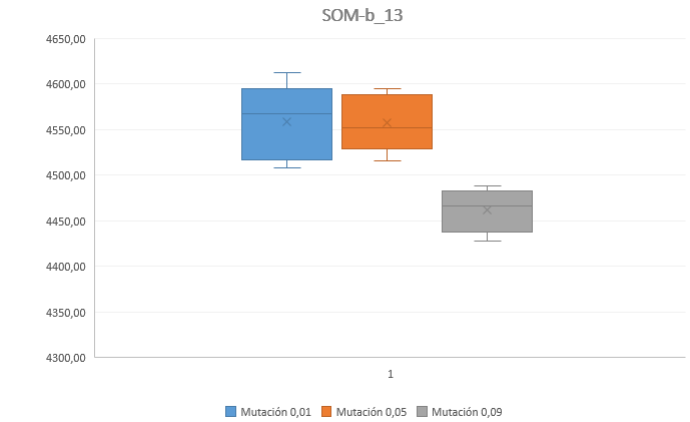


Figure 8: Diferencias en porcentajes respecto a los óptimos globales

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
Mutación 0,01
Mutación 0,05
Mutación 0,09

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