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

```
poblacionPadres ← GeneraPoblacionInicial(semilla)
while evaluacionesRelizadas<evaluacionesLimite do
  Evaluacion(poblacionPadres)
  elites ← SeleccionElites(poblacionPadres)
  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 "poblacionPadres". La función encargada de realizar esta labor es "GeneraPoblacionInicial(semilla)"

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 realizamos dentro del bucle while es calcular los costes de todos y cada uno de los individuos pertenecientes a "poblacionPadres". Esta tarea sera encomendada a la función "Evaluacion(poblacionPadre)".

Cuando ya dispongamos de los costes calculados de todos los individuos de "poblacionPadres", lo siguiente que debemos hacer es guardar en "elites" los individuos élites que nos indique "numElites" (parámetro del programa). La función "SeleccionElites(poblacionPadres)" será la encargada de realizar esta tarea.

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)".

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)

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.

Algorithm 5 CruzarPoblacion(poblacion,semilla)

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

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

A continuación, rellenamos el vector a partir de "poblacion" con la función "SeleccionaPoblacion(poblacion, semilla)".

Una vez tenemos "poblacionHijos" relleno con todos los individuos necesarios, 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.

Algorithm 6 SeleccionaPoblacion(poblacion,semilla)

```
poblacionHijos ← ∅
while tamañoPoblacionHijos<numHijos do
  individuoSeleccionado ← SeleccionaIndividuo(poblacion, semilla)
  poblacionHijos ← poblacionHijos ∪ {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.

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.

Algorithm 8 RealizarCruceMPX(poblacionHijos)

Algorithm 9 RealizarCruce2p(poblacionHijos)

```
for i=0; i<49; i+=2 do
  aleatorioCruce ← GeneraFloatAleatorio(semilla)
  if aleatorioCruce<=probabilidadCruce then
    corte1 ← GeneraEnteroAleatorio(semilla)
    corte2 ← GeneraEnteroAleatorio(semilla)
    padre1 ← poblacionPadre[i]
    padre2 ← poblacionPadre[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[i] ← hijo1
    poblacionHijos[i + 1] ← hijo2
  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.

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
```

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.

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.

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".

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.

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.

Algorithm 15 ReemplazarElite(poblacionHijos,elites)

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

Antes que nada, 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.

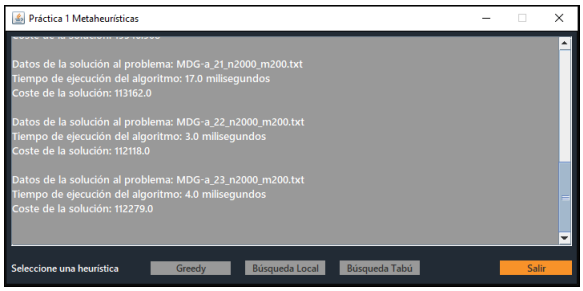

Figure 1: GUI

4.2 Parámetros de los algoritmos

4.2.1 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

References

[1] Fred Glover, Manuel Laguna, Rafael Martí. Principles of Tabu Search. https://www.uv.es/ rmarti/paper/docs/ts1.pdf

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

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