Metaheurísticas: Práctica 2
Técnicas de Búsqueda basadas en Poblaciones para el Problema de la Máxima Diversidad

David Cabezas Berrido

20079906D

Grupo 2: Viernes

dxabezas@correo.ugr.es

7 de mayo de 2021

Índice

1. Descripción y formulación del problema

Nos enfrentamos al **Problema de la Máxima Diversidad** (**Maximum Diversity Problem, MDP**). El problema consiste en seleccionar un subconjunto m elementos de un conjunto de $n > m$ elementos de forma que se **maximice** la *diversidad* entre los elementos escogidos.

Disponemos de una matriz $D = (d_{ij})$ de dimensión $n \times n$ que contiene las distancias entre los elementos, la entrada (i, j) contiene el valor d_{ij}, que corresponde a la distancia entre el elemento i-ésimo y el j-ésimo. Obviamente, la matriz D es simétrica y con diagonal nula.

Existen distintas formas de medir la diversidad, que originan distintas variantes del problema. En nuestro caso, la diversidad será la suma de las distancias entre cada par de elementos seleccionados.

De manera formal, se puede formular el problema de la siguiente forma:

Maximizar

$$f(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j \tag{1}$$

sujeto a

$$\sum_{i=1}^{n} x_i = m$$
$$x_i = \{0, 1\}, \quad \forall i = 1, \ldots, n.$$

Una solución al problema es un vector binario x que indica qué elementos son seleccionados, seleccionamos el elemento i-ésimo si $x_i = 1$.

Sin embargo, esta formulación es poco eficiente y para la mayoría de algoritmos proporcionaremos otra equivalente pero más eficiente.

El problema es **NP-completo** y el tamaño del espacio de soluciones es $\binom{n}{m}$, de modo que es conveniente recurrir al uso de metaheurísticas para atacarlo.

2. Aplicación de los algoritmos

Los algoritmos para resolver este problema tendrán como entradas la matriz D $(n \times n)$ y el valor m. La salida será un contenedor (vector, conjunto, ...) con los índices de los elementos seleccionados, y no un vector binario como el que utilizamos para la formulación. En nuestro caso utilizaremos vectores de enteros para representar soluciones.

Nota: Al contrario de lo recomendado, mantenemos la representación entera (vector de enteros con los elementos seleccionados) en lugar de cambiar a la binaria para las soluciones. Esto conlleva la traducción de los operadores a la nueva representación. Sin embargo, aunque la descripción detallada de los operadores es algo más compleja, entender su funcionamiento es bastante más fácil con la representación entera.

La evaluación de la calidad de una solución se hará sumando la contribución de cada uno de los elementos, y dividiremos la evaluación en dos funciones. En lugar de calcular la función evaluación como en (1), lo haremos así:

$$f(x) = \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} d(i,j) = \frac{1}{2}\sum_{i=1}^{m} \text{contrib}(i) \tag{2}$$

La diferencia es que contamos la distancia entre cada dos elementos i, j dos veces, distancia del elemento i-ésimo al j-ésimo y del j-ésimo al i-ésimo. Esto es obviamente más lento que con $j > i$ en la sumatoria, pero nos permite factorizar la evaluación de la solución como suma de las contribuciones de los elementos, lo cuál será útil para reaprovechar cálculos al evaluar soluciones para la Búsqueda Local. Además, representar la solución como un vector de m índices y no un vector binario de longitud n presenta una clara ventaja: las sumatorias van hasta m en lugar de n. No tenemos que computar distancias para luego multiplicarlas por cero como sugería la formulación en (1).

Presentamos el pseudocódigo de la función para calcular la contribución de un elemento x_i.

Algorithm 1: CONTRIB calcula la contribución de un elemento en una solución.

Input: Un vector de índices S.
Input: La matriz de distancias D.
Input: Un entero e correspondiente al índice del elemento.
Output: La contribución del elemento e, como se describe en (2).

```
contrib ← 0
for s in S do
    contrib ← contrib + D[e, s]        // Sumo las distancias del elemento e a cada elemento de S
return contrib
```

Nótese que el elemento e no tiene que pertenecer al conjunto S. Esto obviamente no ocurrirá cuando se vaya a evaluar una solución al completo invocando esta función con la que describiremos a continuación. Pero, de esta forma, permite conocer cómo influirá en la evaluación el añadir un nuevo elemento sin necesidad de añadirlo realmente.

Ahora presentamos el pseudocódigo de la función para evaluar una solución completa.

Algorithm 2: FITNESS calcula la evaluación de una solución.

Input: Un vector de índices S.
Input: La matriz de distancias D.
Output: El valor de la función objetivo sobre la solución compuesta por S, como se describe en (2).

```
fitness ← 0
for e in S do
    fitness ← fitness + contrib(S, D, e)    // Sumo la contribución de cada elemento de la solución
return fitness/2                            // Hemos contado cada distancia dos veces
```

Podemos definir la distancia de un elemento e a un conjunto S como:

$$d(e,S) = \sum_{s \in S} d(e,s) \tag{3}$$

Esta expresión nos será de utilidad para la implementación de los algoritmos.

Gracias a la existencia del Algoritmo 1, podemos obtener esta expresión como contrib(S, D, e).

En esta práctica, implementamos cuatro variantes de algoritmos genéticos (dos generacionales y dos estacionarios) y tres variantes de meméticos. Compararemos estos algoritmos entre sí y con los algoritmos Greedy y Búsqueda Local con Primer Mejor de la práctica anterior.

Para los pseudocódigos que siguen, suponemos la matriz de distancias D y los parámetros n y m accesibles. El conjunto de todos los elementos es el $\{0, \ldots, n-1\}$, para cuando nos refiramos a elementos de fuera de un subconjunto de ellos.

Inicializamos la población con soluciones aleatorias, usamos la siguiente función.

Algorithm 3: RandomSol proporciona una solución válida aleatoria

Output: Una solución válida del MDP obtenida aleatoriamente.
$E \leftarrow \{0, \ldots, n-1\}$ `// Vector con todos los elementos.`
shuffle(E)
$S \leftarrow \emptyset$ `// La solución empieza vacía.`
while $|S| < m$ **do**
  $S \leftarrow S \cup \{E[|S|]\}$ `// Seleccionamos los` m `primeros elementos de` E`, que son aleatorios.`
return S

2.1. Operadores de los algoritmos genéticos

Usaremos dos operadores de cruce distintos. El primero es el **cruce uniforme**, que dados dos padres mantiene los elementos seleccionados por ambos (intersección) y por ninguno. Los elementos que sólo son seleccionados por uno de los padres se introducen con un 0.5 de probabilidad, pudiendo dar lugar a soluciones con más de m elementos seleccionados. Es por ello que se aplica un **operador de reparación** posteriormente, que elimina (cuando sobran elementos) o añade (cuando faltan) siempre el elemento que más contribuye (dentro o fuera de la solución, según haya que eliminar o añadir).

Cuando escribimos operaciones de conjuntos sobre vectores entendemos que no es relevante el orden de los elementos. Con la unión entre un vector y un elemento, podemos añadir el elemento al final, por ejemplo.

Algorithm 4: Repair repara un vector solución que puede no contener m elementos (puede ser no válida).

Input: Un vector de índices S.
Output: El vector S reparado (no lo devuelve, modifica el existente).
while $|S| > m$ **do**
  $g \leftarrow$ Elemento de S que maximiza contrib(S, D, g)
  $S \leftarrow S \backslash \{g\}$
while $|S| < m$ **do**
  $g \leftarrow$ Elemento de fuera de S que maximiza contrib(S, D, g)
  $S \leftarrow S \cup \{g\}$

A continuación, proporcionamos el pseudocódigo del algoritmo de cruce uniforme.

Algorithm 5: UniformCross genera un hijo cruzando dos padres.

Input: Dos vectores de índices S_1, S_2.
Output: Un vector solución S (hijo).
$S \leftarrow \emptyset$
foreach e *in* $0, \ldots, n-1$ **do**
  if $e \in S_1$ *and* $e \in S_2$ **then**
    $S \leftarrow S \cup \{e\}$
  else if $e \notin S_1$ *and* $e \notin S_2$ **then**
    No se incluye el elemento (no se hace nada).
  else
    Con probabilidad 0.5: $S \leftarrow S \cup \{e\}$
repair(S) `// La solución puede no ser factible.`
return S

El otro operador de cruce que consideramos es el cruce basado en posición. Este operador respeta los seleccionados y descartados por ambos padres, y completa con un subconjunto aleatorio de los elementos que sólo están seleccionados por uno de los padres hasta obtener un vector con m elementos seleccionados.

Algorithm 6: PositionCross genera un hijo cruzando dos padres.

Input: Dos vectores de índices S_1, S_2.
Output: Un vector solución S (hijo).
$S \leftarrow \emptyset$
$W \leftarrow \emptyset$ `// Candidatos a completar la solución.`
foreach e *in* $0, \ldots, n-1$ **do**
  if $e \in S_1$ *and* $e \in S_2$ **then**
    $S \leftarrow S \cup \{e\}$
  else if $e \notin S_1$ *and* $e \notin S_2$ **then**
    No se incluye el elemento (no se hace nada).
  else
    $W \leftarrow W \cup \{e\}$
$W \leftarrow \text{shuffle}(W)$
while $|S| < m$ **do**
  $e \leftarrow W[0]$ `// Primer elemento de` W`, es aleatorio.`
  $S \leftarrow S \cup \{e\}$
  $W \leftarrow W \backslash \{e\}$
return S

Con la implementación que hemos hecho, ambos operadores sólo generan un hijo, por lo que llamaremos a estos operador dos veces cada vez que crucemos dos padres. Sería más eficiente generar dos hijos en cada ejecución para aprovechar parte de los cálculos.

Necesitamos también un operador de mutación. Éste saca un elemento aleatorio de una solución y mete un elemento aleatorio de fuera.

Algorithm 7: Mutate modifica una solución cambiando un elemento.

Input: Un vector solución S.
Output: La solución S modificada, la modifica en lugar de devolverla.
$e_{out} \leftarrow$ numero aleatorio entre 0 y $m-1$ `// Posición del elemento a eliminar.`
$e_{in} \leftarrow$ elemento aleatorio (número aleatorio entre 0 y $n-1$)
while $e_{in} \in S$ **do**
  $e_{in} \leftarrow$ elemento aleatorio `// Forzamos que sea de fuera.`
$S[e_{out}] \leftarrow e_{in}$ `// Sustituimos el elemento a eliminar por el nuevo.`

Por último, necesitamos un operador de selección para elegir a los padres en cada iteración. Se hace uno de torneos binarios, donde se elige el mejor de dos soluciones aleatorias.

Algorithm 8: BinTournament devuelve el índice de la mejor de dos soluciones aleatorias.

Input: Un vector de soluciones P (población).
Output: El índice de la mejor solución entre dos elegidas aleatoriamente.
$i_1 \leftarrow$ número aleatorio entre 0 y $|P|$
$i_2 \leftarrow$ número aleatorio entre 0 y $|P|$
$sol_1 \leftarrow P[i_1]$
$sol_2 \leftarrow P[i_2]$
`/*` $sol.fitness$ `almacena el resultado de` fitness(sol) `por razones de eficiencia. */`
if $sol_1.fitness > sol_2.fitness$ **then**
  return i_1
else
  return i_2

El uso de esta selección para reemplazar la población dependerá del esquema (generacional o estacionario).

2.2. Búsqueda local para los algoritmos meméticos

Proporcionamos la implementación del algoritmo de búsqueda local que usaremos en los algoritmos meméticos. Modifica una solución saltando al primer mejor vecino explorado hasta consumir un cierto número de evaluaciones o alcanzar un máximo local.

Suponemos accesibles las variables globales $LIMIT = 100000$ (límite total de evaluaciones), $EVALS$ (evaluaciones totales hasta el momento, comienza a 0) y $limit = 400$ (límite de evaluaciones en una búsqueda local).

Algorithm 9: LocalSearch modifica una solución con varias iteraciones de búsqueda local con primer mejor.

Input: Solución de partida S.
Output: La solución S se modifica (no se devuelve) con varias iteraciones de búsqueda local.
$E \leftarrow \{0, \ldots, n-1\}$ `// Vector con todos los elementos.`
$evals \leftarrow 0$
$carryon \leftarrow true$
while $carryon$ **do**
  $carryon \leftarrow false$
  $lowest \leftarrow$ indice del elemento de S que menos contribuye, minimiza contrib($S, D, S[lowest]$)
  $E \leftarrow$ shuffle(E) `// Para explorar los posibles vecinos en orden aleatorio.`
  for e ***in*** E **do**
    if $e \in S$ **then**
      continue `// Si ya está escogido, no lo cuento.`
    $contrib \leftarrow$ contrib(S, D, e) $- D[e, S[lowest]]$ `// Contribución a la solución sin el elemento a sustituir.`
    $EVALS \leftarrow EVALS + 1$ `// He evaludado una posible solución.`
    $evals \leftarrow evals + 1$
    if $contrib > min\_contrib$ **then**
      $S.fitness \leftarrow S.fitness + contrib - min\_contrib$ `// Fitness de la nueva solución`
      $carryon \leftarrow true$ `// Toca saltar, lo que completa la iteración`
      $S[lowest] \leftarrow e$ `// Saltamos a la nueva solución`
    if $carryon == true$ *or* $EVALS \geq LIMIT$ *or* $evals \geq limit$ **then**
      break `// Se cumple alguna de las condiciones de parada`

3. Descripción de los algoritmos

Distinguimos dos clases de algoritmos genéticos, según el esquema de reemplazamiento.

3.1. Algoritmo genético generacional

Para seleccionar la nueva población se realizan tantos torneos binarios como el tamaño de la población. Para conservar la mejor solución (elitismo), ésta sustituye a la peor en caso de no sobrevivir a los torneos.

Algorithm 10: REPLACEMENT devuelve la población de padres para la siguiente generación.

Input: Un vector de soluciones P (población).
Output: La población P' de padres para la siguiente generación. No se devuelve, se modifica P.

```
P' ← ∅
best ← índice de la solución de P con mayor fitness
elitism ← false                                   // Para contemplar si sobrevive la mejor.
while |P'| < |P| do
    i ← BinTournament(P)
    P' ← P' ∪ {P[i]}
    if i = best then
        elitism ← true                                                  // Ha sobrevivido.
if elitism = false then
    i ← índice de la solución de P' con peor fitness
    P'[i] ← P[best]
P ← P'
```

Hay que tener en cuenta que comparar si es la mejor solución por índice y no por fitness fuerza a que si hay soluciones repetidas (ocurrirá tras varias iteraciones del algoritmo, cada vez más), se fuerza a salvar una copia concreta de la solución. Esto le da ventaja a la mejor solución respecto a las demás, ya que puede salvarse y además copiarse una vez más. Con esta comparación, se acelera la convergencia del algoritmo pero se reduce la variedad de soluciones, aunque no en gran medida.

Para cruzar la población (de padres), se calcula el número esperado de cruces, $25 \cdot$ probabilidad de cruce $= 18$ (nos quedamos con un entero). El valor 25 proviene del número de parejas que se forman con la población de 50 cromosomas. Como el operador de reemplazamiento construye una nueva población de padres aleatorios, podemos simplemente cruzar primero con segundo, tercero y cuarto, etc. (hasta llegar a 18 cruces). El cruce de dos soluciones puede ser uniforme o posicional, se estudian las dos alternativas.

Algorithm 11: CROSS cruza los padres de la población y los sustituye por los hijos.

Input: Un vector de soluciones P (población).
Output: En la población P, se sustituyen cada pareja de padres por sus hijos.

```
n2cross = n_chromosomes · prob_cross                  // Doble del número esperado de cruces.
for i = 0, 2, 4, ..., n2cross − 1 do
    child_1 ← Croos(P[i], P[i + 1])
    child_2 ← Croos(P[i], P[i + 1])
    P[i] ← child_1
    P[i + 1] ← child_2
```

Para la mutación, se calcula el número esperado de mutaciones. En este caso, el número de mutaciones no depende de n y m, siempre es el 10 % del número de cromosomas, 5. Tantas veces como el número esperado de mutaciones, se elige una solución aleatoria, y esta muta un gen aleatorio como hemos descrito antes. El número de genes coincide con el parámetro n del problema (debemos tener en cuenta que los parámetros que nos proponen corresponden a la representación binaria).

Algorithm 12: MUTATE muta algunas soluciones de la población.

Input: Un vector de soluciones P (población).
Output: En la población P mutan algunas soluciones, no se devuelve nada.
$mutations = n\_chromosomes \cdot n\_genes \cdot prob_{mut}$ // Donde $prob_{mut} = 0.1 \cdot n\_genes$.
for $i = 0, 1, \ldots, mutations - 1$ **do**
  $j \leftarrow$ número aleatorio entre 0 y $|P| - 1$ // Índice de una solución aleatoria.
  mutate(P[j]) // La solución muta un gen aleatorio.

El esquema de reemplazamiento del algoritmo genético generacional es el siguiente.

Algorithm 13: AGG.

Input: Un vector de soluciones P (población) inicializado con soluciones aleatorias.
Output: La población P (se modifica, no se devuelve) evoluciona tras varias generaciones.
while $EVAlS < LIMIT$ **do**
  cross(P)
  mutate(P)
  evaluate(P)
  replacement(P)

En evaluación, se actualiza el fitness de cada solución, que se almacena por razones de eficiencia. Se usa un flag para evitar reevaluar soluciones que no cambien de una generación a otra. El flag está en "actualizado" para nuevas soluciones aleatorias y productos de cruces y mutaciones. Como la comprobación del límite de evaluaciones no se realiza en mitad de las iteraciones, podemos pasarnos del límite. Sin embargo, el número máximo de evaluaciones por iteración es de 41 (36 hijos + 5 mutaciones), por lo que como mucho llegaremos a 100040 evaluaciones, lo que no supone mucho ni en tiempo ni en desempeño del algoritmo.

3.2. Algoritmo genético estacionario

En el esquema estacionario, sólo dos soluciones se cruzan y puden mutar en cada iteración. Los dos padres se eligen con dos torneos binarios.

Algorithm 14: SELECTION devuelve los índices de dos padres, que selecciona por torneo binario.

Input: Un vector de soluciones P (población).
Output: Índices de dos padres.
$p_1 \leftarrow$ BinTournament(P)
$p_2 \leftarrow$ BinTournament(P)
return p_1, p_2

Dichos padres se cruzan para formar dos hijos, por cruce uniforme o basado en posición.

Algorithm 15: CROSS devuelve dos soluciones, producto del cruce de los padres.

Input: Un vector de soluciones P (población).
Input: Los índices de los padres: p_1 y p_2.
Output: Dos nuevas soluciones (hijos).
$child_1 \leftarrow$ cross($P[p_1], P[p_2]$)
$child_2 \leftarrow$ cross($P[p_1], P[p_2]$)
return $child_1, child_2$

Se decide si mutan los hijos (cada uno con probabilidad 0.1, para preservar la esperanza de 0.2 mutaciones por iteración), y posteriormente sustituyen a las dos peores soluciones de la problación (siempre que las superen). De las 4 soluciones (2 peores + 2 hijos), debemos quedarnos con las 2 mejores. Esto lo conseguimos con la siguiente función.

Algorithm 16: REPLACEMENT se queda con las dos mejores de 4 soluciones: 2 peores + 2 hijos.

Input: Un vector de soluciones P (población).
Input: Los dos hijos: $child_1$ y $child_2$.
Output: Modifica la población P para sustituir las peores soluciones por los hijos (si estos las superan).
$w_1, w_2 \leftarrow$ índices de la peor y segunda peor soluciones de P respectivamente
if $child_1.fitness > P[w_2].fitness$ **then**
    $P[w_1] \leftarrow P[w_2]$
    $P[w_2] \leftarrow child_1$
else if $child_1.fitness > P[w_1].fitness$ **then**
    $P[w_1] \leftarrow child_1$
if $child_2.fitness > P[w_1].fitness$ **then**
    $P[w_1] \leftarrow child_2$

El ciclo de evolución queda de la siguiente forma.

Algorithm 17: AGE.

Input: Un vector de soluciones P (población) inicializado con soluciones aleatorias.
Output: La población P (se modifica, no se devuelve) evoluciona tras varias iteraciones.
Se evalúan todas las soluciones (50 evaluaciones), se incrementa $EVALS$ en 50
while $EVAlS < LIMIT$ **do**
    $p_1, p_2 \leftarrow$ selection(P)
    cross(P, p_1, p_2)
    $child_1$ muta con probabilidad 0.1
    $child_2$ muta con probabilidad 0.1
    evaluate$(child_1)$
    evaluate$(child_2)$
    $EVALS \leftarrow EVALS + 2$
    replacement$(P, child_1, child_2)$

3.3. Algoritmos meméticos

Los algoritmos meméticos combinan el esquema evolutivo generacional con el algoritmo de búsqueda local. Ya hemos mostrado la implementación de búsqueda local a nivel de solución. Ahora mostraremos cómo se aplica la búsqueda local a nivel de población, lo que diferencia las distintas variantes de meméticos.

En la primera versión, AM-(10,1.0), se aplica búsqueda local cada 10 generaciones sobre todos los cromosomas de la población.

Algorithm 18: LOCALSEARCH de AM-(10,1.0).

Input: Un vector de soluciones P (población).
Output: La población P (se modifica, no se devuelve) después de que todas las soluciones mejoren con búsqueda local.
foreach *sol in P* **do**
    LocalSearch(*sol*)

En la versión AM-(10,0.1), se aplica búsqueda local a cada cromosoma con probabilidad 0.1 cada 10 generaciones.

Algorithm 19: LOCALSEARCH de AM-(10,0.1).

Input: Un vector de soluciones P (población).
Output: La población P (se modifica, no se devuelve) después de que algunas de las soluciones mejoren con búsqueda local.
foreach *sol in P* **do**
    Con probabilidad 0.1: LocalSearch(*sol*)

En la versión AM-(10,0.1mej), se aplica búsqueda local a las 5 mejores soluciones (mejor 10 % de la población).

Algorithm 20: LocalSearch de AM-(10,0.1mej).

Input: Un vector de soluciones P (población).
Output: La población P (se modifica, no se devuelve) después de que las 5 mejores soluciones mejoren con búsqueda local.

$q \leftarrow \emptyset$ `// Cola con prioridad donde almacenaré parejas (fitness,índice), se ordenan por mayor fitness.`
foreach $i = 0, \ldots, |P| - 1$ **do**
    $q \leftarrow q \cup \{(P[i].fitness, i)\}$
$k \leftarrow 0.1 \cdot n\_chromosomes$ `// Sacaré los 10% mejores (5).`
for $j = 0, \ldots, k-1$ **do**
    LocalSearch($P[q.top.second]$)
    $q.pop$

El cuerpo principal de los algoritmos meméticos es el siguiente, la función LocalSearch (a nivel de población) es lo que diferencia las distintas alternativas.

Algorithm 21: AM.

Input: Un vector de soluciones P (población) inicializado con soluciones aleatorias.
Output: La población P (se modifica, no se devuelve) evoluciona tras varias iteraciones.
Se evalúan todas las soluciones (50 evaluaciones), se incrementa $EVALS$ en 50
$generation \leftarrow 0$
while $EVAlS < LIMIT$ **do**
    cross(P)
    mutate(P)
    evaluate(P)
    replacement(P)
    $generation \leftarrow generation + 1$
    if $generation \bmod 10 == 0$ **then**
        LocalSearch(P)

4. Desarrollo de la práctica

La implementación de los algoritmos y la experimentación con los mismos se ha llevado acabo de C++, utilizando la librería STL. Para representar la soluciones hemos hecho uso del tipo `vector`.

La mayoría de operadores (mutación, cruce, búsqueda local) se implementan a nivel de solución y a nivel de población para abstraer las operaciones y que el código sea más reciclable, generalmente como métodos de clase.

Para medir los tiempos de ejecución se utiliza la función `clock` de la librería `time.h`.

A lo largo de la práctica se utilizan acciones aleatorias. Utilizamos la librería `stdlib.h` para la generación de enteros (no negativos) pseudoaleatorios con `rand` y fijamos la semilla con `srand`. Se barajan vectores con la función `random_shuffle` de la librería `algorithm`.

Para las acciones que se realizan con cierta probabilidad, es necesario generar flotantes pseudoaleatorios en el intervalos $[0, 1]$. Para esto, se genera un entero no negativo con `rand` y se divide entre el máximo posible (RAND_MAX).

Se almacena la matriz de distancias completa (no sólo un triángulo) por comodidad de los cálculos.

Se utiliza optimización de código `-O2` al compilar.