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

11 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 (algoritmos implementados en esta práctica) 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 (AGG)

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 (AGE)

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 (AM)

Los algoritmos meméticos combinan el esquema evolutivo generacional (con cruce uniforme) 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)

3.4. Búsqueda local

Procedemos con la descripción del algoritmo de Búsqueda Local que se nos ha presentado en el seminario. Este algoritmo utiliza la técnica del Primer Mejor, en la que se van generando soluciones en el entorno de la actual y se salta a la primera con mejor evaluación. Para la implementación del algoritmo, necesitamos distintos elementos.

El primer elemento, es una función para generar una solución aleatoria de partida. Simplemente se eligen m elementos diferentes del conjunto. Por comodidad, también calculamos el complementario.

Algorithm 22: RandomSol proporciona una solución válida aleatoria

Input: El entero m.
Input: El entero n.
Output: Una solución válida del MDP obtenida aleatoriamente.
Output: El complementario de la solución obtenida.

```
E ← {0, ..., n − 1}                          // Conjunto con los elementos no seleccionados
S ← ∅                                        // La solución empieza vacía
while |S| < m do
    e ← elemento aleatorio de E
    E ← E\{e}
    S ← S ∪ {e}
return S
return E                                     // El complementario
```

Lo siguiente que necesitamos es un método para generar las soluciones del entorno. Estas soluciones se consiguen sustituyendo el menor contribuyente de la solución actual por otro candidato. Presentamos el código para obtener el menor contribuyente.

Algorithm 23: lowestContrib obtiene el elemento de S que menos contribuye en la valoración.

Input: Un conjunto de elementos S.
Input: La matriz de distancias D.
Output: El elemento de S que minimiza contrib(S, S, e) con $e \in S$.
Output: Su contribución, para la factorización de la función objetivo.

```
lowest ← primer elemento de S
min_contrib ← contrib(S, D, lowest)
for s in S do
    contrib ← contrib(S, D, s)
    if contrib < min_contrib then
        min_contrib ← contrib
        lowest ← s                  // Si encuentro un candidato con menor contribución, actualizo
return lowest
return min_contrib
```

En el caso de que S se represente como un conjunto, no sabemos cuál será el primer elemento (depende de la implementación del iterador). Pero esto no es relevante, ya que vale cualquier elemento de S.

Finalmente, proporcionamos el algoritmo de Búsqueda Local para actualizar la solución por otra del entorno iterativamente hasta encontrar un máximo local (una solución mejor que todas las de su entorno) o llegar a un límite de evaluaciones de la función objetivo: $LIMIT = 100000$. Las soluciones del entorno se generan aleatoriamente.

Algorithm 24: LocalSearch

Input: El entero m.
Input: La matriz de distancias D, $n \times n$.
Output: Una solución válida del MDP por el algoritmo de BS que hemos descrito, junto con su evaluación.

```
S ← randomSol(m, n)                          // Comenzamos con una solución aleatoria
E ← {0, ..., n − 1}\S                        // randomSol también devuelve el complementario
fitness ← fitness(S)                         // Diversidad de la solución
E ← vector(E)                                // No importa el orden, pero debe poder barajarse
carryon ← true
LIMIT ← 100000                               // Límite de llamadas a la función de evaluación
CALLS ← 0
while carryon do
    carryon ← false
    lowest = lowestContributor(S, D)
    min_contrib ← contrib(S, D, lowest)      // Se calcula dentro de lowestContributor
    S ← S\{lowest}
    E ← shuffle(E)
    for e in E do
        contrib ← contrib(S, D, e)
        CALLS ← CALLS + 1                    // He evaludado una posible solución
        if contrib > min_contrib then
            fitness ← fitness + contrib − min_contrib   // Diversidad de la nueva solución
            carryon ← true                   // Toca saltar, lo que completa la iteración
            S ← S ∪ {e}                      // Saltamos a la nueva solución
            E ← E\{e}
            E ← E ∪ {lowest}
        if carryon == true or CALLS ≥ LIMIT then
            break                            // Se cumple alguna de las condiciones de parada
if |S| < m then
    S ← S ∪ {lowest}     // Si salimos porque no encontramos una mejor, recuperamos la solución
return S
return fitness
```

Cabe destacar que en este algoritmo se calcula la fitness factorizando. Esto acelera mucho los cálculos, ya que hay que evaluar muchas soluciones diferentes.

4. Algoritmo de comparación: Greedy

Para comparar la eficacia de cada algoritmos, lo compararemos con el algoritmo **Greedy**. El algoritmo consiste en empezar por el elemento más lejano al resto e ir añadiendo el elemento que más contribuya hasta completar una solución válida.

Como elemento más lejano al resto se toma el elemento cuya suma de las distancias al resto sea la mayor. Y en cada iteración se introduce el elemento cuya suma de las distancias a los seleccionados sea mayor. Es decir, utilizamos la definición de (3).

Para calcular ambos valores, usamos la siguiente función, que permite obtener el de entre un conjunto de candidatos más lejano (en el sentido que acabamos de comentar) a los elementos de un conjunto dado. El código para calcularlo es similar al del algoritmo 23.

Algorithm 25: FARTHEST obtiene el candidato más lejano a los elementos de S.

Input: Un conjunto de candidatos C.
Input: Un conjunto de elementos S.
Input: La matriz de distancias D.
Output: El candidato más lejano en el sentido de (3).

```
farthest ← primer elemento de C
max_contrib ← contrib(S, D, farthest)
for e in C do
    contrib ← contrib(S, D, e)
    if contrib > max_contrib then
        max_contrib ← contrib
        farthest ← e                    // Si encuentro un candidato con mayor contribución, actualizo
return farthest
```

En el caso de que C se represente como un conjunto, no sabemos cuál será el primer elemento (depende de la implementación del iterador). Pero esto no es relevante, ya que vale cualquier elemento de C.

Ya estamos en condiciones de proporcionar una descripción del algoritmo Greedy.

Algorithm 26: GREEDY

Input: La matriz de distancias D.
Input: El entero m.
Output: Una solución válida del MDP obtenida como hemos descrito anteriormente, y su diversidad.

```
C ← {0, ..., n − 1}                          // En principio los n elementos son candidatos
S ← ∅                                                    // La solución empieza vacía
farthest ← farthest(C, C, D)                          // Elemento más lejano al resto
C ← C\{farthest}
S ← S ∪ {farthest}
while |S| < m do
    farthest ← farthest(C, S, D)                // Elemento más lejano a los seleccionados
    C ← C\{farthest}
    S ← S ∪ {farthest}
return S
return fitness(S)
```

5. 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.

5.1. Manual de usuario

A continuación detallamos instrucciones para lanzar los ejecutables.

Tenemos los siguientes ejecutables:

- **AGG-uniforme:** Implementación del algoritmo genético generacional con cruce uniforme.
- **AGG-posicion:** Implementación del algoritmo genético generacional con cruce basado en posición.
- **AGE-uniforme:** Implementación del algoritmo genético estacionario con cruce uniforme.
- **AGE-posicion:** Implementación del algoritmo genético estacionario con cruce basado en posición.
- **AM-10-1:** Implementación del algoritmo memético (genético generacional con cruce uniforme combinado con búsqueda local) que aplica búsqueda local a todos los cromosomas cada 10 iteraciones.
- **AM-10-01:** Implementación del algoritmo memético que aplica búsqueda local a cada cromosoma con probabilidad 0.1 cada 10 iteraciones.
- **AM-10-01mej:** Implementación del algoritmo memético que aplica búsqueda local a los "0.1 $\times$ número de cromosomas" mejores cromosomas cada 10 iteraciones.

Todos ellos devuelven la evaluación de la solución obtenida y el tiempo de ejecución por salida estándar. Leen el fichero por entrada estándar, así que es conveniente redirigirla. Todos los archivos reciben la semilla como parámetro.

Además, todos los archivos de búsqueda local reciben la semilla como parámetro. Ejemplo:

```
bin/AGG-uniforme 197 < datos/MDG-a_1_n500_m50.txt >> salidas/AGG-uniforme.txt
```

En la carpeta **software** se incluye el script usado para lanzar todas las ejecuciones, `run.sh`. También se incluye el `Makefile` que compila los ejecutables.

6. Experimentación y análisis

Toda la experimentación se realiza en mi ordenador portátil personal, que tiene las siguientes especificaciones:

- OS: Ubuntu 20.04.2 LTS x86_64.
- RAM: 8GB, DDR4.
- CPU: Intel Core i7-6700HQ, 2.60Hz.

6.1. Casos de estudio y resultados

Tratamos varios casos con distintos parámetros n y m. En cada caso se utiliza una semilla diferente, pero se usa la misma para todos los algoritmos. A continuación presentamos una tabla con los casos estudiados. Para cada caso indicamos los valores de n y m y la semilla que se utiliza.

Ahora mostraremos para cada algoritmo una tabla con los estadísticos (Desviación y Tiempo) que han obtenido en cada caso.

Greedy

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7610.42	2.85	0.001375
MDG-a_2_n500_m50	7574.39	2.54	0.001293
MDG-a_3_n500_m50	7535.96	2.88	0.001304
MDG-a_4_n500_m50	7551.52	2.81	0.001281
MDG-a_5_n500_m50	7540.14	2.77	0.001284
MDG-a_6_n500_m50	7623.65	1.93	0.001278
MDG-a_7_n500_m50	7594.62	2.28	0.0014
MDG-a_8_n500_m50	7625.94	1.61	0.001367
MDG-a_9_n500_m50	7547.25	2.87	0.001351
MDG-a_10_n500_m50	7642.27	1.77	0.001893
MDG-b_21_n2000_m200	11099332.620328	1.77	0.319017
MDG-b_22_n2000_m200	11149879.733826	1.21	0.313017
MDG-b_23_n2000_m200	11119613.974858	1.6	0.303374
MDG-b_24_n2000_m200	11106996.970212	1.63	0.311278
MDG-b_25_n2000_m200	11114220.292214	1.61	0.306411
MDG-b_26_n2000_m200	11132801.799043	1.41	0.306542
MDG-b_27_n2000_m200	11130608.965587	1.55	0.310595
MDG-b_28_n2000_m200	11110673.520354	1.5	0.318429
MDG-b_29_n2000_m200	11156328.082493	1.25	0.306362
MDG-b_30_n2000_m200	11109767.818822	1.65	0.296905
MDG-c_1_n3000_m300	24617010	1.07	1.501668
MDG-c_2_n3000_m300	24547293	1.44	1.464132
MDG-c_8_n3000_m400	43056071	0.88	2.546235
MDG-c_9_n3000_m400	42958639	1.1	2.569214
MDG-c_10_n3000_m400	42959794	1.19	2.566065
MDG-c_13_n3000_m500	66493045	0.78	3.67213
MDG-c_14_n3000_m500	66449858	0.79	3.767131
MDG-c_15_n3000_m500	66468837	0.78	3.78725
MDG-c_19_n3000_m600	94929882	0.74	5.183856
MDG-c_20_n3000_m600	94979205	0.69	5.582157

Tabla 1: Evaluación de las soluciones y estadísticos *Desv* y *Tiempo* obtenidos por el algoritmo Greedy en cada caso de estudio.

Media de los estadísticos:

Desv	Tiempo (s)
1.63	1.19

Búsqueda local

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7623.23	2.69	0.001809
MDG-a_2_n500_m50	7590.18	2.34	0.001391
MDG-a_3_n500_m50	7544.94	2.76	0.001204
MDG-a_4_n500_m50	7576.44	2.49	0.0012
MDG-a_5_n500_m50	7484.27	3.49	0.001308
MDG-a_6_n500_m50	7570.96	2.61	0.001297
MDG-a_7_n500_m50	7654.98	1.5	0.001608
MDG-a_8_n500_m50	7623.78	1.64	0.002379
MDG-a_9_n500_m50	7612.74	2.02	0.001494
MDG-a_10_n500_m50	7619.52	2.07	0.001959
MDG-b_21_n2000_m200	11181874.0007	1.04	0.099777
MDG-b_22_n2000_m200	11167876.184	1.05	0.092492
MDG-b_23_n2000_m200	11176568.0611	1.09	0.107634
MDG-b_24_n2000_m200	11188223.318	0.91	0.107425
MDG-b_25_n2000_m200	11181859.8196	1.01	0.090053
MDG-b_26_n2000_m200	11193478.832	0.88	0.122694
MDG-b_27_n2000_m200	11211629.6839	0.83	0.112468
MDG-b_28_n2000_m200	11151089.4629	1.14	0.079449
MDG-b_29_n2000_m200	11183039.6644	1.01	0.09833
MDG-b_30_n2000_m200	11159590.8213	1.21	0.090033
MDG-c_1_n3000_m300	24729057	0.62	0.601221
MDG-c_2_n3000_m300	24738675	0.67	0.584432
MDG-c_8_n3000_m400	43200330	0.55	1.264437
MDG-c_9_n3000_m400	43157977	0.64	1.241837
MDG-c_10_n3000_m400	43188306	0.66	1.195051
MDG-c_13_n3000_m500	66636142	0.56	2.304507
MDG-c_14_n3000_m500	66727635	0.38	2.430114
MDG-c_15_n3000_m500	66808383	0.28	2.78715
MDG-c_19_n3000_m600	95244690	0.41	3.572005
MDG-c_20_n3000_m600	95324379	0.33	3.598978

Tabla 2: Evaluación de las soluciones y estadísticos *Desv* y *Tiempo* obtenidos por el algoritmo de Búsqueda Local con Primer Mejor en cada caso de estudio.

Media de los estadísticos:

Desv	Tiempo (s)
1.3	0.69

AGG con cruce uniforme

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7707.95	1.61	2.090093
MDG-a_2_n500_m50	7638.38	1.71	2.107509
MDG-a_3_n500_m50	7606.73	1.97	2.000252
MDG-a_4_n500_m50	7572.09	2.55	1.9783
MDG-a_5_n500_m50	7578.8	2.27	2.056744
MDG-a_6_n500_m50	7595.96	2.29	2.003538
MDG-a_7_n500_m50	7657.02	1.48	2.028599
MDG-a_8_n500_m50	7573.56	2.29	1.996006
MDG-a_9_n500_m50	7624.78	1.87	2.047315
MDG-a_10_n500_m50	7559.2	2.84	2.024727
MDG-b_21_n2000_m200	11121197.565411	1.58	47.359572
MDG-b_22_n2000_m200	11140356.96411	1.3	46.470377
MDG-b_23_n2000_m200	11122989.485668	1.57	47.038308
MDG-b_24_n2000_m200	11093069.977321	1.75	47.933936
MDG-b_25_n2000_m200	11160806.463414	1.2	46.681582
MDG-b_26_n2000_m200	11128960.812991	1.45	46.708763
MDG-b_27_n2000_m200	11151190.815364	1.37	46.649915
MDG-b_28_n2000_m200	11133921.949524	1.29	44.099666
MDG-b_29_n2000_m200	11105148.429167	1.7	43.791342
MDG-b_30_n2000_m200	11115025.068757	1.61	44.419215
MDG-c_1_n3000_m300	24606743	1.11	142.839721
MDG-c_2_n3000_m300	24604945	1.21	142.000313
MDG-c_8_n3000_m400	43018354	0.96	201.906448
MDG-c_9_n3000_m400	43010797	0.98	199.613779
MDG-c_10_n3000_m400	42986441	1.13	203.129084
MDG-c_13_n3000_m500	66338876	1.01	244.070216
MDG-c_14_n3000_m500	66332815	0.97	242.688824
MDG-c_15_n3000_m500	66450927	0.81	248.337499
MDG-c_19_n3000_m600	94857517	0.81	293.797158
MDG-c_20_n3000_m600	94808873	0.87	303.61342

Tabla 3: Evaluación de las soluciones y estadísticos *Desv* y *Tiempo* obtenidos por el algoritmo AGG con cruce uniforme en cada caso de estudio.

Media de los estadísticos:

Desv	Tiempo (s)
1.52	90.12

AGG con cruce basado en posición

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7524.33	3.95	1.930193
MDG-a_2_n500_m50	7568.98	2.61	1.939586
MDG-a_3_n500_m50	7541.52	2.81	1.945953
MDG-a_4_n500_m50	7521.62	3.2	1.923407
MDG-a_5_n500_m50	7539.26	2.78	1.952541
MDG-a_6_n500_m50	7538.96	3.02	1.938942
MDG-a_7_n500_m50	7450.6	4.13	1.937036
MDG-a_8_n500_m50	7564.71	2.4	1.939857
MDG-a_9_n500_m50	7603.57	2.14	1.930254
MDG-a_10_n500_m50	7544.26	3.03	2.016837
MDG-b_21_n2000_m200	10992775.091951	2.72	27.771752
MDG-b_22_n2000_m200	10976004.218225	2.75	27.515051
MDG-b_23_n2000_m200	11010381.091843	2.56	27.593846
MDG-b_24_n2000_m200	11004690.027815	2.53	27.324046
MDG-b_25_n2000_m200	11050334.301811	2.18	27.377645
MDG-b_26_n2000_m200	11009755.0324	2.5	27.681516
MDG-b_27_n2000_m200	10973671.631925	2.94	27.388455
MDG-b_28_n2000_m200	11007899.484688	2.41	27.441754
MDG-b_29_n2000_m200	10965254.29057	2.94	27.687189
MDG-b_30_n2000_m200	10969766.811796	2.89	27.704564
MDG-c_1_n3000_m300	24234790	2.61	81.964096
MDG-c_2_n3000_m300	24291889	2.46	81.279382
MDG-c_8_n3000_m400	42565620	2.01	122.55461
MDG-c_9_n3000_m400	42575671	1.98	124.047167
MDG-c_10_n3000_m400	42429806	2.41	125.035444
MDG-c_13_n3000_m500	65814139	1.79	157.365497
MDG-c_14_n3000_m500	65837578	1.71	158.397447
MDG-c_15_n3000_m500	65882485	1.66	157.807931
MDG-c_19_n3000_m600	94069575	1.64	193.966365
MDG-c_20_n3000_m600	94140624	1.57	192.747059

Tabla 4: Evaluación de las soluciones y estadísticos *Desv* y *Tiempo* obtenidos por el algoritmo AGG con cruce basado en posición en cada caso de estudio.

Media de los estadísticos:

Desv	Tiempo (s)
2.54	56.34

AGE con cruce uniforme

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7699.87	1.71	1.935585
MDG-a_2_n500_m50	7577.32	2.5	1.79422
MDG-a_3_n500_m50	7539.43	2.83	1.799771
MDG-a_4_n500_m50	7527.6	3.12	1.755444
MDG-a_5_n500_m50	7492.01	3.39	1.743837
MDG-a_6_n500_m50	7604.02	2.18	1.792661
MDG-a_7_n500_m50	7509.91	3.37	1.791742
MDG-a_8_n500_m50	7564.46	2.41	1.74758
MDG-a_9_n500_m50	7567.89	2.6	1.818509
MDG-a_10_n500_m50	7579.83	2.58	1.940848
MDG-b_21_n2000_m200	11096803.204221	1.8	28.572898
MDG-b_22_n2000_m200	11066582.864213	1.95	30.746676
MDG-b_23_n2000_m200	11103929.892324	1.73	28.877216
MDG-b_24_n2000_m200	11108833.598636	1.61	29.766053
MDG-b_25_n2000_m200	11089928.650404	1.82	30.116922
MDG-b_26_n2000_m200	11072601.605505	1.95	28.99646
MDG-b_27_n2000_m200	11098217.011779	1.84	31.245306
MDG-b_28_n2000_m200	11091400.95864	1.67	31.786776
MDG-b_29_n2000_m200	11056917.76965	2.13	29.736697
MDG-b_30_n2000_m200	11098912.065773	1.75	31.97064
MDG-c_1_n3000_m300	24515388	1.48	98.19262
MDG-c_2_n3000_m300	24514080	1.57	94.704585
MDG-c_8_n3000_m400	42758112	1.56	136.57157
MDG-c_9_n3000_m400	42837534	1.38	134.870618
MDG-c_10_n3000_m400	42870141	1.39	139.704011
MDG-c_13_n3000_m500	66172152	1.26	180.704192
MDG-c_14_n3000_m500	66320235	0.98	170.5091
MDG-c_15_n3000_m500	66331255	0.99	175.780283
MDG-c_19_n3000_m600	94735806	0.94	214.486399
MDG-c_20_n3000_m600	94753449	0.93	220.617744

Tabla 5: Evaluación de las soluciones y estadísticos *Desv* y *Tiempo* obtenidos por el algoritmo AGE con cruce uniforme en cada caso de estudio.

Media de los estadísticos:

Desv	Tiempo (s)
1.91	62.87

AGE con cruce basado en posición

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7499.7	4.27	1.925836
MDG-a_2_n500_m50	7507.96	3.39	1.943304
MDG-a_3_n500_m50	7537.63	2.86	1.957983
MDG-a_4_n500_m50	7646.03	1.6	1.926846
MDG-a_5_n500_m50	7577.33	2.29	1.942608
MDG-a_6_n500_m50	7533.03	3.1	1.934613
MDG-a_7_n500_m50	7627.8	1.85	1.925517
MDG-a_8_n500_m50	7621.37	1.67	1.929454
MDG-a_9_n500_m50	7581.98	2.42	1.937019
MDG-a_10_n500_m50	7591.91	2.42	1.988036
MDG-b_21_n2000_m200	10971888.149454	2.9	28.643575
MDG-b_22_n2000_m200	11027565.453894	2.3	28.229621
MDG-b_23_n2000_m200	10984557.930365	2.79	28.259755
MDG-b_24_n2000_m200	11019573.82926	2.4	28.206778
MDG-b_25_n2000_m200	11031909.811213	2.34	28.188983
MDG-b_26_n2000_m200	10972375.024121	2.83	28.235294
MDG-b_27_n2000_m200	11019671.669981	2.53	29.779949
MDG-b_28_n2000_m200	11003349.24807	2.45	28.305121
MDG-b_29_n2000_m200	10984303.49364	2.77	28.312105
MDG-b_30_n2000_m200	11018900.201837	2.46	28.268545
MDG-c_1_n3000_m300	24257830	2.52	79.378184
MDG-c_2_n3000_m300	24288993	2.47	80.020735
MDG-c_8_n3000_m400	42566989	2	122.195934
MDG-c_9_n3000_m400	42606289	1.91	123.499277
MDG-c_10_n3000_m400	42579055	2.06	123.454948
MDG-c_13_n3000_m500	65940834	1.6	162.591331
MDG-c_14_n3000_m500	65888399	1.63	160.209596
MDG-c_15_n3000_m500	65949859	1.56	159.929673
MDG-c_19_n3000_m600	94208266	1.49	198.198681
MDG-c_20_n3000_m600	94167554	1.54	198.774549

Tabla 6: Evaluación de las soluciones y estadísticos *Desv* y *Tiempo* obtenidos por el algoritmo AGE con cruce basado en posición en cada caso de estudio.

Media de los estadísticos:

Desv	Tiempo (s)
2.35	57.07

AM-(10,1.0)

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7700.82	1.7	0.185877
MDG-a_2_n500_m50	7721.71	0.64	0.163316
MDG-a_3_n500_m50	7705.94	0.69	0.19582
MDG-a_4_n500_m50	7645.81	1.6	0.203958
MDG-a_5_n500_m50	7643.15	1.45	0.210678
MDG-a_6_n500_m50	7664.54	1.4	0.186335
MDG-a_7_n500_m50	7700.58	0.92	0.180075
MDG-a_8_n500_m50	7692.35	0.76	0.202284
MDG-a_9_n500_m50	7623.02	1.89	0.174959
MDG-a_10_n500_m50	7711.32	0.89	0.165259
MDG-b_21_n2000_m200	11098034.367619	1.79	12.714229
MDG-b_22_n2000_m200	11114872.596709	1.52	13.74357
MDG-b_23_n2000_m200	11046514.713112	2.24	13.221175
MDG-b_24_n2000_m200	11085811.030799	1.82	12.426276
MDG-b_25_n2000_m200	11108364.164213	1.66	13.379022
MDG-b_26_n2000_m200	11101324.660648	1.69	12.462323
MDG-b_27_n2000_m200	11108922.583065	1.74	12.367685
MDG-b_28_n2000_m200	11114670.65925	1.46	12.96751
MDG-b_29_n2000_m200	11163653.99827	1.18	12.191321
MDG-b_30_n2000_m200	11092297.742468	1.81	11.631424
MDG-c_1_n3000_m300	24501042	1.54	41.494163
MDG-c_2_n3000_m300	24455245	1.81	39.206223
MDG-c_8_n3000_m400	42827259	1.4	52.130071
MDG-c_9_n3000_m400	42743253	1.6	50.978821
MDG-c_10_n3000_m400	42808199	1.54	51.95662
MDG-c_13_n3000_m500	66183870	1.24	70.166402
MDG-c_14_n3000_m500	66120270	1.28	67.915979
MDG-c_15_n3000_m500	66236107	1.13	65.384416
MDG-c_19_n3000_m600	94745444	0.93	79.415658
MDG-c_20_n3000_m600	94389441	1.31	89.668006

Tabla 7: Evaluación de las soluciones y estadísticos *Desv* y *Tiempo* obtenidos por el algoritmo AM-(10,1.0).

Media de los estadísticos:

Desv	Tiempo (s)
1.42	24.58

AM-(10,0.1)

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7665.34	2.15	0.444765
MDG-a_2_n500_m50	7638.93	1.71	0.424028
MDG-a_3_n500_m50	7659.83	1.28	0.427798
MDG-a_4_n500_m50	7594.19	2.27	0.454077
MDG-a_5_n500_m50	7644.02	1.43	0.423357
MDG-a_6_n500_m50	7602.98	2.2	0.43389
MDG-a_7_n500_m50	7622.97	1.91	0.425276
MDG-a_8_n500_m50	7646.34	1.35	0.409405
MDG-a_9_n500_m50	7644.58	1.62	0.408827
MDG-a_10_n500_m50	7623.36	2.02	0.415836
MDG-b_21_n2000_m200	11154224.651148	1.29	24.167352
MDG-b_22_n2000_m200	11195682.50995	0.81	25.845168
MDG-b_23_n2000_m200	11193129.056159	0.95	23.585301
MDG-b_24_n2000_m200	11154194.151774	1.21	27.342935
MDG-b_25_n2000_m200	11181857.898937	1.01	23.255834
MDG-b_26_n2000_m200	11185064.252571	0.95	24.836796
MDG-b_27_n2000_m200	11186293.835698	1.06	23.695422
MDG-b_28_n2000_m200	11136103.80939	1.27	21.437087
MDG-b_29_n2000_m200	11164170.189057	1.18	23.421367
MDG-b_30_n2000_m200	11154631.751513	1.26	24.846379
MDG-c_1_n3000_m300	24728126	0.63	84.572393
MDG-c_2_n3000_m300	24697000	0.84	72.995771
MDG-c_8_n3000_m400	43165013	0.63	100.799762
MDG-c_9_n3000_m400	43149724	0.66	105.070718
MDG-c_10_n3000_m400	43187178	0.66	100.059467
MDG-c_13_n3000_m500	66773925	0.36	122.149366
MDG-c_14_n3000_m500	66647248	0.5	119.359096
MDG-c_15_n3000_m500	66732655	0.39	116.756599
MDG-c_19_n3000_m600	95201709	0.45	140.959524
MDG-c_20_n3000_m600	95187705	0.48	141.718517

Tabla 8: Evaluación de las soluciones y estadísticos *Desv* y *Tiempo* obtenidos por el algoritmo AM-(10,0.1).

Media de los estadísticos:

Desv	Tiempo (s)
1.15	45.04

AM-(10,0.1mej)

Caso	Coste obtenido	Desv	Tiempo (s)
MDG-a_1_n500_m50	7637.16	2.51	0.414972
MDG-a_2_n500_m50	7724.21	0.61	0.41606
MDG-a_3_n500_m50	7650.23	1.41	0.411984
MDG-a_4_n500_m50	7619.15	1.94	0.431361
MDG-a_5_n500_m50	7683.11	0.93	0.415595
MDG-a_6_n500_m50	7606.02	2.16	0.406575
MDG-a_7_n500_m50	7671.13	1.29	0.407201
MDG-a_8_n500_m50	7640.97	1.42	0.417129
MDG-a_9_n500_m50	7623.97	1.88	0.392046
MDG-a_10_n500_m50	7609.15	2.2	0.387585
MDG-b_21_n2000_m200	11132452.586865	1.48	19.528066
MDG-b_22_n2000_m200	11189058.525244	0.87	23.840448
MDG-b_23_n2000_m200	11199499.507476	0.89	25.008253
MDG-b_24_n2000_m200	11153274.975023	1.22	25.542597
MDG-b_25_n2000_m200	11197661.105297	0.87	25.065803
MDG-b_26_n2000_m200	11175094.576497	1.04	22.660221
MDG-b_27_n2000_m200	11214002.699731	0.81	24.63223
MDG-b_28_n2000_m200	11190418.039251	0.79	23.46206
MDG-b_29_n2000_m200	11181245.238434	1.03	20.418433
MDG-b_30_n2000_m200	11165943.182085	1.15	21.691616
MDG-c_1_n3000_m300	24680269	0.82	68.57299
MDG-c_2_n3000_m300	24667185	0.96	74.06677
MDG-c_8_n3000_m400	43190089	0.57	86.488724
MDG-c_9_n3000_m400	43125997	0.72	98.001809
MDG-c_10_n3000_m400	43136622	0.78	88.114333
MDG-c_13_n3000_m500	66704966	0.46	112.764369
MDG-c_14_n3000_m500	66664685	0.47	110.111094
MDG-c_15_n3000_m500	66655997	0.5	109.479115
MDG-c_19_n3000_m600	95292197	0.36	130.071368
MDG-c_20_n3000_m600	95132559	0.53	121.509782

Tabla 9: Evaluación de las soluciones y estadísticos *Desv* y *Tiempo* obtenidos por el algoritmo AM-(10,0.1mej).

Media de los estadísticos:

Desv	Tiempo (s)
1.09	41.17

Comparamos los estadísticos medios obtenidos estos algoritmos entre sí y con los obtenidos por los algoritmos de búsqueda local (con primer mejor) y greedy de la práctica anterior.

Algoritmo	Desv	Tiempo (s)
Greedy	1.63	1.19
BL	1.3	0.69
AGG-uniforme	1.52	90.12
AGG-posicion	2.54	56.34
AGE-uniforme	1.91	62.87
AGE-posicion	2.35	57.07
AM-(10,1.0)	1.42	24.58
AM-(10,0.1)	1.15	45.04
AM-(10,0.1mej)	1.09	41.17

Tabla 10: Comparativa de los estadísticos medios obtenidos por los distintos algoritmos.

6.2. Análisis de resultados

A la vista de la Tabla 10, intuimos que todos son algoritmos decentes para la resolución de este problema, aunque hay diferencias entre ellos.

Sin duda, los tiempos de los algoritmos que sólo manejan una solución (búsqueda local y greedy) en lugar de una población de soluciones son mucho más rápidos. El tiempo de cómputo de estos algoritmos tiene un orden de magnitud bastante inferior a los algoritmos basados en poblaciones. En cuanto a fitness, los algoritmos de la práctica 1 obtienen mejores resultados (en general) que los genéticos, pero se ven superados por los meméticos, que obtienen los mejores resultados entre las alternativas que consideramos.

A grandes rasgos: algoritmos genéticos frente a meméticos

Respecto a la fitness conseguida, la ninguno de los **algoritmos genéticos** logra superar a la búsqueda local, y sólo el AGG-uniforme supera (por poco) a greedy. Esto posiblemente se deba a la falta de explotación en estos algoritmos, ya que no buscan explícitamente soluciones mejores, sino esperan que se generen por medio de cruces y mutaciones y se mantengan por la simulación que hacemos de selección natural.

El problema de los algoritmos genéticos es que cuando la mejor solución es replicada en muchos de los cromosomas, algo que acaba pasando debido al comportamiento del algoritmo y a lo que llamamos "convergencia del algoritmo", es muy difícil que el algoritmo encuentre nuevas soluciones. Esto se debe a que el operador de cruce de dos soluciones idénticas genera hijos idénticos a los padres. Por tanto, es posible que rápidamente toda la población se vea "invadida" por una o varias soluciones mejores que el resto, y esto lleve a una pérdida de diversidad (me refiero a diversidad genética, variedad entre soluciones) significativa. Por tanto, es posible que estos algoritmos converjan (encuentren una solución buena) rápidamente y después se pasen un gran número de iteraciones trabajando con soluciones muy similares hasta que una mutación o cruce fortuito mejore la fitness de la población, dando lugar a un gran número de evaluaciones desperdiciadas.

Para justificar esta hipótesis, resaltamos que la función que calcula los índices de la peor y segunda peor solución de la población en el esquema estacionario (se invoca en el Algoritmo 16) acaba devolviendo 0 y 1 a partir de un número de iteraciones, y rara vez vuelve a devolver valores distintos. Esto quiere decir que encontrar soluciones mejores es muy difícil para el algoritmo a partir de cierto punto, y se limita a agotar las evaluaciones restantes con la esperanza de que alguna mutación o cruce origine una solución mejor.

Podríamos pensar que nuestra implementación del reemplazamiento elitista en AGG puede ser causante de este problema (ver comentarios tras Algoritmo 10), pero esto no explica el problema en el caso de AGE, que de hecho obtiene soluciones peores en media (ligeramente mejores para el cruce basado en posición, pero peores para el cruce uniforme). Por tanto nuestra implementación del elitismo como mucho acentúa ligeramente este problema, que es intrínseco a los algoritmos genéticos.

Para solventar esto, añadimos un componente de explotación como es la búsqueda local, dando lugar a los **algoritmos meméticos** (implementados sobre el esquema generacional con cruce uniforme, el AGG que mejores resultados obtiene). De esta forma, una parte de las evaluaciones se dedica a la exploración (genéticos) y otra a la explotación

(búsqueda local). Este equilibrio los convierte en los más adecuados para el problema entre las alternativas consideradas, ya que consiguen las mejores fitness con la excepción del AM-(10,1.0), que es ligeramente superado por la búsqueda local.

Además, estos algoritmos consiguen un efecto que combate el problema de la pérdida de la convergencia demasiado rápida en los algoritmos genéticos que hemos comentado anteriormente. Aplicar la búsqueda local a algunas soluciones separa los miembros de la población unos de otros y da lugar a un conjunto más variado de soluciones. Por ejemplos, si el algoritmo encuentra una buena solución que acaba copiándose multiples veces y desbancando al resto, aplicar búsqueda local a algunas de las copias de esa solución provocarán que el conjunto de soluciones sea más variado, con lo que podemos esperar cruces de soluciones más distintas y que alguno de ellos dé lugar a una solución mejor. Esta mejora se nota menos en AM-(10,1.0), ya que aplica búsqueda local a todos los cromosomas y no sólo a algunos, de hecho esta variante de memético consigue los peores resultados entre este tipo de algoritmo. El hecho de que la búsqueda local se realice con primer mejor y explorando los entornos en orden aleatorio, consigue que soluciones iguales se transformen en soluciones distintas, debido a que la aleatoriedad de exploración del entorno provoca que no se salte siempre al mismo vecino (al contrario que en la búsqueda local con mejor), esta diversificación ocurre también aunque se aplique búsqueda local a todas las soluciones y no sólo a algunas, luego también mejora AM-(10,1.0). Ésto sin mencionar que la alternaiva del primer mejor consume muchísimas menos evaluaciones que la búsqueda local con mejor, lo que comprobamos como trabajo voluntario en la anterior práctica.

En resumen, los algoritmos genéticos carecen de explotación y pueden tener el problema de que se pierda la diversidad demasiado pronto, y la introducción de la búsqueda local combate ambos problemas.

Respecto al tiempo, los algoritmos meméticos tardan mucho menos, ya que en la búsqueda local se dedica mucho más porcentaje del tiempo de cómputo a evaluar soluciones. Además, estas evaluaciones son mucho más rápidas al calcularse la fitness de forma factorizada.

Cruce uniforme frente a cruce basado en posición

El operador de cruce uniforme proporciona mejores resultados, lo que puede deberse a que palia esa pérdida de diversidad genética que hemos comentado. A pesar de que parecen muy similares (los dos respetan las decisiones tomadas por ambos padres), existe una diferencia que hace mejor al cruce uniforme.

Primero, ambos cruces eligen y descartan los elementos en los que ambos padres estén de acuerdo. Después se rellena la solución de forma aleatoria en ambos casos, con la siguiente diferencia: la probabilidad de que un elemento sólo seleccionado por uno de los padres sea seleccionado por el hijo es siempre de $\frac{\text{número de huecos por cubrir}}{\text{número de elementos seleccionados por sólo un padre}}$ en el cruce basado en posición, mientras que en el cruce uniforme esta probabilidad depende de la aportación del elemento. En cruce uniforme se introduce cada uno de los elementos con probabilidad 0.5, y a partir de ahí se elimina o se añade el que más aporta dependiendo de si faltan o sobran elementos. De esta forma, algunos hijos llevan un componente de explotación que recuerda al greedy (cuando faltan elementos y se añaden los que más aportan), mientras que otros hijos se empeoran a propósito para no quedarnos sin variedad en las soluciones (cuando sobran elementos y se eliminan los que más aportan). Este comportamiento sugiere que los hijos procedentes de cruces uniformes serán más variados que los basados en posición, y además habrá algunos hijos (en los que se añadan elementos de fuera) que tendrán cierta componente de explotación similar a la del greedy, lo que podría solventar parcialmente los problemas de los algoritmos genéticos que comentamos con anterioridad.

Como consecuencia usamos el cruce uniforme (y el esquema generacional) para los algoritmos meméticos.

Respecto al tiempo, el algoritmo de reparación puede requerir varias búsquedas del elemento que más aporta, que son bastante costosas, sobre todo cuando vienen de fuera de la solución. Esto provoca que sea más lento, se aprecia más en el esquema generacional.

AGG frente a AGE

No hay una gran diferencia en los resultados obtendios por ambos esquemas de reemplazamiento. El esquema generacional ha resultado algo más efectivo en media debido a su mejor desempeño con el cruce uniforme. Respecto a tiempos, el esquema generacional es algo más lento cuando se utiliza el cruce uniforme.

Variantes de meméticos

Además de añadir una componente de explotación de la que carecen los algoritmos genéticos. Hemos explicado cómo la introducción de la búsqueda local (sobre todo con primer mejor) aumenta la diversidad entre los cromosomas, es más notable en las variantes en las que sólo se aplica a algunos de los cromosomas. Como consecuencia, los algoritmos meméticos proporcionan (en en media) los mejores resultados

Dentro de los los meméticos, AM-(10,0.1) y AM-(10,0.1mej) ofrecen un mejor desempeño debido a esa mayor diversidad genética en las soluciones y a que no compensa invertir demasiadas itereaciones en mejorar todos los cromosomas, ya que cierto número de soluciones no sobreviven cada generación.

Dentro de sólo aplicar la búsqueda local a algunas soluciones, obtiene ligeramente mejores resultados (en media) la alternativa que la aplica a las mejores (a las 10 % mejores), probablemente debido a que aprovechan mejor la explotación. Mejoran aun más las soluciones con más fitness, lo que al cabo de varias generaciones se traduce en una mayor mejora de la población. Puesto que la búsqueda local sólo se realiza hasta consumir 400 evaluaciones por cromosoma, las mejores más altas tienen más posibilidades de alcanzar una mayor fitness.

Respecto al tiempo, AM-(10,1.0) es el memético más rápido con diferencia, ya que dedica 10 veces más que las otras variantes a hacer búsqueda local, que es una forma mucho más rápida de consumir evaluaciones por la cantidad que hace y por hacerlas factorizadas.