Problema de Máxima Diversidad (MDP)

Técnicas basadas en poblaciones

Metaheurísticas: Práctica 2, Grupo 1

José Antonio Álvarez Ocete - 77553417Q
joseantonioao@correo.ugr.es

June 8, 2020

Contents

1 Introducción: práctica 2

De cara a la segunda práctica se han añadido a la memoria las secciones 3.2, 4.2, 5.2 y 6.2. El resto permanece invariante.

2 El problema

2.1 Descripción del problema

El **problema de la máxima diversidad** (en inglés, *maximum diversity problem*, MDP) es un problema de optimización combinatoria que consiste en seleccionar un subconjunto M de m elementos ($|M| = m$) de un conjunto inicial N de n elementos (con $n > m$) de forma que se maximice la diversidad entre los elementos escogidos. El MDP se puede formular como:

$$\text{Maximizar } z_{MS}(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} \cdot x_i \cdot x_j$$

$$\text{Sujeto a } \sum_{i=1}^{n} x_i = m$$

$$x_i \in \{0,1\}, \forall i \in \{1,\ldots,n\}$$

Donde:

- x es una solución al problema que consiste en un vector binario que indica los m elementos seleccionados.
- d_{ij} es la distancia existente entre los elementos i y j.

2.2 Casos considerados

Se utilizarán 30 casos seleccionados de varios de los conjuntos de instancias disponibles en la *MDPLIB* (`http://www.optsicom.es/mdp/`), 10 pertenecientes al grupo **GKD** con distancias Euclideas, $n = 500$ y $m = 50$ ($GKD - c\_i\_n500\_m50$ para $i \in \{1,\ldots,10\}$), 10 del grupo **SOM** con distancias enteras entre 0 y 999, $n \in \{300, 400, 500\}$ y $m =\in \{40,\ldots,200\}$ ($SOM - b\_11\_n300\_m90$ a $SOM - b\_20\_n500\_m200$) y 10 del grupo **MDG** con distancias enteras entre 0 y 10, $n = 2000$ y $m = 200$ ($MDG - a\_i\_n2000\_m200$ para $i \in \{21,\ldots,30\}$.

Puesto que la numeración utilizada es unívoca se hará referencia a estas entradas simplemente como **MDP-a_i** con $i \in \{1,\ldots,10\}$, **SOM-b_i** con $i \in \{11,\ldots,20\}$ y **GKD-c_i** con $i \in \{21,\ldots,30\}$.

3 Descripción de la aplicación de los algoritmos

En esta sección describiremos las consideraciones comunes a los distintos algoritmos. Este incluye la representación de las soluciones, la función objetivo y los operadores comunes a los distintos algoritmos. Ya que los únicos puntos en común de la búsqueda local y la técnica greedy son la función objetivo y la representación de las soluciones no estudiaremos ningún operador común. Tampoco se han incluido los detalles específicos de un algoritmo a pesar de que sean comunes a varios algoritmos finales, ya que estos son pequeñas variaciones unos y otros.

El lenguaje utilizado para la implementación de la práctica ha sido $C++$.

3.1 Práctica 1

3.1.1 Representación de la soluciones

El esquema de representación de una solución es el siguiente:

```
struct solution {
        vector<int> v;
        double fitness;
};
```

Donde el vector contiene números entre 1 y n no repetidos que componen la solución ($|v| = m$). Aunque el orden de estos elementos no es relevante se utilizará determinado ordenamiento sobre este mismo vector en algunas de las soluciones planteadas.

Cabe destacar que los datos proporcionados únicamente representan las distancias punto a punto, para todos los puntos. Sin embargo se desconoce la posición exacta de cada elemento. Es por esto por lo que no se ha podido implementar la técnica Greedy planteada inicialmente ya que se centraba en el concepto de *centroide* o *baricentro* de un conjunto y no podíamos calcularlo sin estimar primero la posición de los puntos.

Estos datos se han almacenado en una matriz simétrica de tamaño n x n que de aquí en adelante denotaremos por MAT.

3.1.2 Función objetivo

Para la función objetivo se ha dividido la implementación en dos funciones ya que algunos algoritmos utilizarán unicamente una de las dos y otros, ambas. La primera calcula la contribución del elemento i a la solución actual. La segunda calcula el $fitness$ total utilizando la función anterior.

3.2 Práctica 2

3.2.1 Representación de la soluciones

De cara a trabajar con técnicas basadas en poblaciones se ha implementado una segunda representación de las soluciones basada en *booleanos* en vez de enteros. Se almacena además un *booleano* adicional para saber si la solución esta actualmente evaluada, así como su valor *fitness* Esto representará un cromosoma de nuestra población.

```
struct solution {
        vector<bool> v;
        double fitness;
        bool evaluated;
};
```

Se ha utilizado también una clase para almacenar una población. Esta encapsula además de los cromosomas, la posición actual del mejor cromosoma y su valor *fitness*. Esto será especialmente útil en los algoritmos genéticos generacionales y meméticos, que no mantendrán la población ordenada sino unicamente cuál es el mejor cromosoma.

```
class population {
public:
        vector<solution> v;
        double max_fitness;
        int best_sol;
        int tam;
```

```
        [...]
    };
```

Debido a que la implementación de las técnicas locales utiliza el esquema de representación de enteros se han implementado también sendos algoritmos de transformación de una representación a otra. La implementación de los mismo es trivial y por ello no se incluye en esta memoria.

Cabe destacar que aunque ambas representaciones se llamen de la misma forma, se usan en contextos distintos y no hay ambigüedad posible. En el único caso en el que no es así (los algoritmos meméticos), la representación que utilizada enteros se denominará **solution_int**.

3.2.2 Función objetivo

Se ha desarrollado otra función objetivo análoga a la implementada para la representación con enteros que utiliza de nuevo la función *SingleContribution*, reimplementando también ésta para la codificación con *booleanos*.

[H] v : $vector < bool >$, elem: int result : double result ← 0
i ∈ $[0, v.size)$ result ← v[i] * MAT[i][elem]

[H] sol : solution fitness : double sol.fitness ← 0
i ∈ $[0, sol.v.size)$ sol.v[i] sol.fitness ← SingleContribution (sol , i)
// Counting twice all the possible distances
sol.fitness ← sol.fitness / 2
sol.evaluated ← true

3.2.3 Operadores

Para esta práctica se han implementado dos operadores distintos de cruce y dos de reemplazamiento. Los operadores de mutación, selección (por torneo binario) e inicialización aleatoria son compartidos por todos los algoritmos. Procedemos a estudiar cada operador por separado. En primer lugar, el operador de selección utilizado ha sido torneo binario. Para ello utilizamos la función *random(a,b)*, que devuelve un entero aleatorio en el intervalo $[a, b)$.

[H] pop : population elem : int r1 ← random(0, pop.tam)
r2 ← random(0, pop.tam)
pop.v[r1].fitness > pop.v[r2].fitness elem ← r1 elem ← r2

Para el operador de mutación simplemente se cambia uno de los valores elegidos por otro que no este escogido ya, de forma aleatoria:

[H] pop : population elem : int r1 ← random(0, pop.tam)
r2 ← random(0, pop.tam)
!$sol.v[r\_on]$ r_on ← random(0, sol.v.size()) !$sol.v[r\_off]$ OR r_on == r_off r_off ← random(0, sol.v.size()) sol.v[r_on] ← false
sol.v[r_off] ← true

sol.evaluated oldContribution ← singleContribution(sol.v, r_on) - MAT[r_on][r_off]
NewContribution ← singleContribution(sol.v, r_off)
sol.fitness ← sol.fitness + newContribution - oldContribution
TOTAL_EVALUATIONS ← TOTAL_EVALUATIONS + 1

En este algoritmos hay un par de detalles adicionales que explicar. Por un lado en el último bloque *if* utilizamos una evaluación de la solución factorizada equivalente a la de la práctica 1. Esta es aplicable unicamente cuando la solución estaba correctamente evaluada antes de la mutación. Adicionalmente consideramos **TOTAL_EVALUATIONS** como el número de

evaluaciones totales del algoritmo actual. Cuando evaluamos la solución, aumentamos en 1 el número de evaluaciones.

Debido a las características de la aproximación al problema esta mejora se verá reflejada unicamente en el tiempo, ya que a pesar de haber reducido la evaluación de $O(n^2)$ a $O(n)$, contamos la evaluación como una completa.

Pasamos a explicar los operadores de cruce. En primer lugar se ha implementado un cruce uniforme así como un operador de reparación. Por un lado el cruce uniforme mantiene en los hijos las elecciones (eligir o no elegir un elemento) comunes de los dos padres, y asigna elecciones aleatorias para los demás valores. Acto seguido se aplica el operador de reparación para que los hijos sean soluciones válidas. Es decir, escojan exactamente m elementos.

[H] p1 : solution, p2 : solution child : solution child ← p1
child.evaluated ← false
m ← 0
$i \in [0, p1.v.size())$ p1.v[i] m ← m + 1 p1.v[i] AND p2.v[i] child.v[i] ← true !p1.v[i] AND !p2.v[i] child.v[i] ← false child.v[i] ← random(0,2) == 0
child ← repairSolution(child, m)

[H] child : solution, m : int child : solution nTrues ← countTrues(child.v)
nTrues < m r ← random(0, sol.v.size())
sol.v[r] sol.v[r] ← false nTrues ← nTrues + 1 nTrues > m r ← random(0, sol.v.size())
!sol.v[r] sol.v[r] ← true nTrues ← nTrues - 1

El algoritmo de reparación utilizado ha sido este en lugar del que aparece en las transparencias por la enorme carga de trabajo que suponía el presentado en clase ($O(n^2)$).

En segundo lugar presentamos el operador de cruce posicional. Este las elecciones positivas comunes entre los padres y asigna un reordenamiento del resto de elementos a los no asignados.

[H] p1 : solution, p2 : solution c1 : solution, c2 : solution c1 ← p1
c2 ← p2
c1.evaluated ← false
c2.evaluated ← false
shuffled = vector¡int¿ (size = c1.v.size(), value=false)
$i \in [0, p1.v.size())$ p1.v[i] AND p2.v[i] c1.v[i] ← true
c2.v[i] ← true
shuffled[i] ← true
to_shuffle[i] ← p1.v[i]
to_shuffle1 ← randomShuffle(to_shuffle)
to_shuffle2 ← randomShuffle(to_shuffle)
$i \in [0, p1.v.size())$ shuffled[i] c1.v[i] ← to_shuffle1[i]
c2.v[i] ← to_shuffle2[i]

Finalmente detallamos los algoritmos de reemplazamiento. Para el generacional elitista basta con sustituir la población completa por la nueva y, en caso de que nuestra mejor solución empeore, sustituir una solución de la nueva generación por la mejor de la generación anterior.

Para el estacionario unicamente generaremos dos hijos por generación. El reemplazamiento se realiza introduciendo estos dos descendientes en la población, la ordenamos en función de su fitness y la truncamos para mantener el número de cromosomas invariante.

[H] pop : population, child1 : solution, child2 : solution pop : population pop.v.append(child1)
pop.v.append(child2)

pop.v ← sort(pop.v)
pop.v ← pop.v.resize(pop.v.size() - 2)

4 Algoritmos

4.1 Práctica 1

Para esta práctica se han implementado un total de 4 algoritmos que a continuación describiremos en profundidad. Son los siguientes:

- Greedy
- Búsqueda local (BL)
- Búsqueda local determinista (BLD)
- Búsqueda local con greedy ($BLcongreedy$)

4.1.1 Greedy

Como ya se ha comentado, la concepción inicial de la práctica era utilizar la idea del *baricentro*. Para ello en cada iteración se calcularía el baricentro de los elementos de la solución y se escogería el punto más alejado a este entre los aún no escogidos. Intentando simular esta estrategia utilizaremos la suma de las distancias al resto de elementos de la solución para cada punto aún no he escogido.

En primer lugar se ha implementado una función que, dados dos conjuntos de puntos, *seleccionados* y *no_seleccionados*, devuelve el elemento de *no_seleccionados* cuya suma de distancias a los elementos de *seleccionados* es máxima. Para ello utilizamos la función *SingleContribution* explicado en el apartado anterior, que computa la suma de distancias de un punto a otro conjunto dado.

[H] selected : $set < int >$, non_selected : $set < int >$ farthest : i fitness ← 0
max_sum_dist ← 0
i ∈ non_selected current_sum_dist ← SingleContribution (selected , i)
current_sum_dist > max_sum_dist max_sum_dist ← current_sum_dist
farthest ← i

De cara al algoritmo greedy final necesitaremos inicializar el conjunto de elementos seleccionados con al menos un elemento. Este será el que esté más alejado de todos los demás. Para calcularlo utilizamos una abstracción de la función anterior, donde *selected* y *non_selected* serán ambos el conjunto de todos los puntos posibles: $\{0, 1, \ldots, n\}$.

[H] none farthest : i all_elements ← $\{0, 1, \ldots, n\}$
farthest ← farthestToSel(all_elements, all_elements)

Finalmente presentamos el algoritmo greedy al completo, haciendo uso de las funciones anteriores.

[H] none selected : $set < int >$ non_selected ← $\{0, 1, \ldots, n\}$
selected ← { farthestToAll() }
$|selected| < m$ farthest ← farthestToSel(selected, non_selected)
non_selected ← non_selected $\bigcup$ {farthest}
selected ← selected - {farthest}

4.1.2 Búsqueda local

Desgranemos la búsqueda local paso por paso. En primer lugar generamos una solución aleatoria que será el punto de partida. En cada iteración exploramos el vecindario hasta encontrar una solución mejor y sustituimos la actual por la encontrada. Repetimos este proceso hasta que la exploración del vecindario no encuentre una solución mejor.

[H] none sol : solution sol ← randomSolution()
stop ← false
!stop stop , sol ← stepInNeighbourhood(sol)

La exploración del vecindario realizada en la función *stepInNeighbourhood* tiene dos detalles relevantes a explicar. Por un lado se ha explicado la factorización del movimiento en el vecindario. Esto es, para estudiar si una solución es mejor en vez de calcular la fitness de la nueva solución y compararla con la actual, estudiamos si el intercambio del elemento $i \in sol$ y el elemento j de los no seleccionados tiene una repercusión positiva en la fitness de la solución. Para ello hacemos uso de la función *SingleContribution* comparando las contribuciones de cada elemento. Si el intercambio merece la pena ($SingleContribution(i, sol) < SingleContribution(j, sol)$) reajustamos la fitness sumandole la diferencia entre ambas.

Por otro lado, a la hora de escoger que elementos i, j comparar, seleccionamos un elemento j que aún no esté en la solución de forma aleatoria y comparamos con todos los posibles $i \in sol$. Una pequeña mejora consiste en ordenar los elementos de la solución en función a la contribución que realizan a esta para intentar intercambiar primero los que menos contribuyan. Veamos como está implementada esta ordenación. Por un lado definimos un operador de comparación que trabaje con parejas $< int, double >$. Esto nos permitirá ordenar el vector solución en orden de contribución creciente.

[H] p1 : $pair < int, double >$, p2 : $pair < int, double >$ comp : bool comp ← $p1.second < p2.second$

A continuación reordenamos los elementos de la solución.

[H] sol : solution sol : solution pairs : $vector < pair < int, double >>$
$i \in sol.v$ pairs[i].first ← i
pairs[i].second ← singleContribution(sol.v, i); pairs ← sort(pairs, *operator* <)
$i \in sol.v$ sol.v[i] ← pairs[i].first

Para acabar presentamos la exploración del vecindario, donde $rand(a, b)$ devuelve un número entero aleatorio en $[a, b)$. Se ordena el vector solución utilizando *orderSolutionByContribution* y se toman $i \in sol.v$ en orden creciente y $j \notin sol.v$ para el intercambio. Este j se tomará de forma aleatoria y con el objetivo de explotar plenamente la ordenación utilizada generaremos múltiples j's para cada i.

Según el guión de prácticas hemos de generar hasta $MAX = 50.000$ vecinos (parejas (i, j) en nuestro caso) antes de parar la búsqueda local. Tras varias pruebas he decidido que merece la pena centrarse en el 10% más prometedor de la solución. Llamemos a este porcentaje *percentage_studied*.

Por lo tanto estudiaremos $max_i = percentage\_studied \cdot |sol|$ elementos de la solución y para cada uno generaremos $max\_random = MAX/max\_i$ valores aleatorios distintos, obteniendo un total de $max\_i \cdot max\_random = max\_i \cdot (MAX/max\_i) = MAX$ elementos en total.

[H] sol : solution stop : bool , sol : solution sol ← orderSolutionByContribution(sol)
$max\_i \leftarrow percentage\_studied \cdot |sol|$

$max\_randoms \leftarrow MAX/max\_i$
stop $\leftarrow$ true
tries $\leftarrow 0$
i $\leftarrow 0$
$i < max\_i$ element_out $\leftarrow$ sol.v[i]
oldContribution $\leftarrow$ singleContribution(sol.v, element_out)
j $\leftarrow rand(0, m)$
k $\leftarrow 0$
$k < max\_k \quad j \notin sol.v$ newContribution $\leftarrow singleContribution(sol.v, j) - MAT[j][element\_out]$
$newContribution > oldContribution$ sol.v[i] $\leftarrow$ j
sol.fitness $\leftarrow sol.fitness + newContribution - oldContribution$
pairs[i].first $\leftarrow$ i
sol $\leftarrow$ false
return $k \leftarrow k + 1$
j $\leftarrow rand(0, m)$
$i \leftarrow k + 1$

4.1.3 Búsqueda local determinista

Este algoritmo esta basado en la búsqueda local recién explicada pero con una pequeña mejora. A la hora de explorar el vecindario también ordenaremos los elementos no seleccionados en función de los más prometedores. Para ello utilizamos la función *obtainBestOrdering*.

[H] sol : solution best_ordering : vector¡int¿ pairs : $vector < pair < int, double >>$
$i \in 0, \ldots, n \quad i \notin sol.v$ pairs[i].first $\leftarrow$ i
pairs[i].second $\leftarrow$ singleContribution(sol.v, i); pairs $\leftarrow$ sort(pairs, $operator <$)
i from $|pairs| - 1$ to 0 best_ordering[i] $\leftarrow$ pairs[i].first

Para el algoritmo en si, repetiremos un razonamiento análogo al anterior. Fijado el número de elementos del vecindario a explorar, MAX, exploramos un porcentaje p_i de la solución y un porcentaje p_k del ordenamiento generado a partir de la función *obtainBestOrdering*.

Recorreremos la solución hasta $max_i = |sol| \cdot p_i$ y los posibles intercambios hasta $max_k = n \cdot p_k$, teniendo en cuenta que:

$$max_k \cdot max_i = (n \cdot p_k) \cdot (|sol| \cdot p_i) = MAX$$

Si damos un valor a p_i podemos calcular max_i y max_k en función del resto de datos: $max_k = MAX/max_i$. Finalmente tenemos que tener en cuenta la posibilidad de que max_k sea mayor que el tamaño de $best_ordering$ es por esto que tomamos $max_k = min(MAX/max_i, |best_ordering|)$. En ese caso hemos de actualizar max_i a $min(MAX/max_k, |sol|)$ para asegurarnos de que hacemos la MAX exploraciones.

Este movimiento esta implementado en la función *stepInNeighbourhoodDet* que a su vez es llamado por el algoritmo final, *localSearchDet*.

[H] sol : solution stop : bool , sol : solution sol $\leftarrow$ orderSolutionByContribution(sol)
best_ordering $\leftarrow$ obtainBestOrdering(sol)
$percent_i \leftarrow 0.1$ $max_i \leftarrow percent_i \cdot |sol|$
$max_k \leftarrow min(MAX/max_i, |best\_ordering|)$
$max_k == |best\_ordering|$ $max_i \leftarrow min(MAX/max_k, |sol|)$ stop $\leftarrow$ true
i $\leftarrow 0$

$i < max\_i$ element_out ← sol.v[i]
oldContribution ← singleContribution(sol.v, element_out)
k ← 0
$k < max\_k$ j ← best_ordering[k] $j \notin sol.v$ newContribution ← $singleContribution(sol.v, j) - MAT[j][element\_out]$
$newContribution > oldContribution$ sol.v[i] ← j
sol.fitness ← $sol.fitness + newContribution - oldContribution$
pairs[i].first ← i
sol ← false
return $k \leftarrow k + 1$
$i \leftarrow k + 1$

[H] none sol : solution sol ← randomSolution()
stop ← false
!stop stop , sol ← stepInNeighbourhoodDet(sol)

Cabe destacar que este algoritmo no es determinista por completo ya que la solución inicial tomada es puramente aleatoria. Este detalle es importante puesto que por ello merecerá la pena ejecutarlo múltiples veces en vez de una única vez.

4.1.4 Búsqueda local con greedy

El último algoritmo presentado es otra mejora a la búsqueda local. Consiste simplemente en tomar como solución inicial la obtenida por el greedy.

[H] none sol : solution sol ← greedy()
stop ← false
!stop stop , sol ← stepInNeighbourhoodDet(sol)

4.2 Práctica 2

Para esta práctica se han implementado un total de 4 algoritmos que a continuación describiremos en profundidad. Son los siguientes:

- Algoritmo Genético Generacional con cruce uniforme - **AGGu**
- Algoritmo Genético Generacional con cruce posicional - **AGGp**
- Algoritmo Genético Estacionario con cruce uniforme - **AGEu**
- Algoritmo Genético Estacionario con cruce posicional - **AGEp**
- Algoritmo Memético - **AM**

En la siguiente tabla se resume la elección de operador para cada algoritmo. Recordemos que los operadores de mutación y y selección por torneo binario son compartidos entre todos los algoritmos.

	Cruce Uniforme	Cruce Positional	Reemplazo Generacional Elitista	Reemplazo Estacionario
AGGp		X	X	
AGGu	X		X	
AGEp		X		X
AGEu	X			X
AM		X	X	

Table 1: Selección de operador por algoritmo

4.2.1 Algoritmos Genéticos Generacionales - AGG

Como su propio nombre indica utilizan técnicas generacionales y elitistas, forzando mantener al mejor elemento de la población en cada generación. Se utiliza por tanto el operador de reemplazamiento generacional. La estructura del algoritmo es la siguiente:

[H] nChoosen : int, MAX_EVALUATIONS : int best_sol : solution , generations : int
generations ← 0
pop ← initializePop(tam_pob)
pop, TOTAL_EVALUATIONS ← evaluatePop(pop)

TOTAL_EVALUATIONS < MAX_EVALUATIONS new_pop ← selection(pop)
new_pop ← crossPop(new_pop)
new_pop, TOTAL_EVALUATIONS ← mutatePop(new_pop)
new_pop, TOTAL_EVALUATIONS ← evaluatePop(new_pop)
new_pop ← replace(pop, new_pop)
generations ← generations + 1
best_sol ← pop.v[pop.best_sol]

Podemos apreciar en el código algunos parámetros que aún no tienen valor, como el número máximo de evaluaciones ($MAX\_EVALUATIONS$), que utilizaremos de condición de parada, o las probabilidades de cruce ($cross\_prob$) y mutación (mut_prob). Daremos valores a estos parámetros en las sección 5.2.

Como ya se ha comentado, hay dos algoritmos distintos que utilizarán el código recién mostrado: **AGGu** y **AGGp**. La única diferencia entre ambos será el operador de cruce que utilicen: uniforme y posicional respectivamente.

4.2.2 Algoritmos Genéticos Estacionario - AGE

A continuación detallamos los algoritmos genéticos de tipo estacionario que en cada iteración únicamente seleccionan dos padres que, tras ser cruzados y mutados, compiten por entrar de nuevo en población. Su algoritmo en pseudo-código es el siguiente:

[H] nChoosen : int, MAX_EVALUATIONS : int best_sol : solution , generations : int
generations ← 0
pop ← initializePop(tam_pob)
pop, TOTAL_EVALUATIONS ← evaluatePop(pop)

TOTAL_EVALUATIONS < MAX_EVALUATIONS p1, p2 ← selectPair(pop)
c1, c2 ← crossPair(p1, p2)
c1, c2 ← mutatePair(c1, c2)
c1 ← evaluateSolution(c1)

c2 ← evaluateSolution(c2)
TOTAL_EVALUATIONS ← TOTAL_EVALUATIONS + 2
pop ← replace(pop, c1, c2)
generations ← generations + 1
best_sol ← pop.v[pop.best_sol]

De nuevo tendremos dos algoritmos distintos según el tipo de operador de cruce que se utilice, uniforme o posicional: **AGEu** y **AGEp**.

4.2.3 Algoritmo Meméticos - AM

Los algoritmos meméticos utilizan el esquema evolutivo de reemplazo generacional aplicando una búsqueda local a ciertos cromosomas cada cierto número de generaciones. Se puede aplicar una búsqueda más exhaustiva en intervalos generacionales amplios o pequeñas búsquedas para intervalos más cortos. En nuestro caso utilizaremos aproximaciones distintas:

- Aplicar la búsqueda local a toda la población cada 10 iteraciones - **AM0**
- Aplicar la búsqueda local a elementos arbitrariamente escogidos cada 10 iteraciones - **AM1**
- Aplicar la búsqueda local al mejor cromosoma de la población cada 10 iteraciones - **AM2**

Este esquema queda reflejado en la función **memetize**, dependiendo del valor de *mem_type*:
[H] pop : population, MAX_ITERATIONS : int, p_mem : double, mem_type : int pop-ulation : pop mem_type == 0 $sol \in pop.v$ sol ← localSearch(sol, MAX_ITERATIONS) mem_type == 1 $sol \in pop.v$ rand() < p_mem sol ← localSearch(sol, MAX_ITERATIONS) pop.v[pop.best_sol] ← localSearch(pop.best_sol, MAX_ITERATIONS)
En cada llamada a *localSearch* fijamos un valor de iteraciones máximo, normalmente reducido para que el tiempo de ejecución no se dispare, y todas las evaluaciones en dicha subllamada se añaden al valor total de $TOTAL\_ITERATIONS$, teniéndolas en cuenta para el criterio de parada del algoritmo completo. Cabe destacar que tras tomar ciertos datos decidí que la evaluación factorizada de una solución contase unicamente como una fracción de una evaluación total, en vez de como una completa. Es decir, aumento el valor de $TOTAL\_ITERATIONS$ cuando evalúo tantas soluciones como elementos escogidos en el vector. Tomé esta decisión debido porque apenas exploraba el entorno de la solución y no obtenía mejora sustancial. Esta misma técnica de evaluación será aplicada en los próximos algoritmos que aún no hemos explicado.

Por otro lado, el pseudo-código del caso 1 ha sido simplificado. En vez de generar *pop.tam* números aleatorios calculamos $p\_mem \cdot pop.tam$ y mutamos ese número de cromosomas de igual forma que al mutar una población completa. Si el valor $p\_mem \cdot pop.tam$ es más pequeño que uno, se genera un único número aleatorio en $[0, 1]$ y se muta un cromosoma si es más pequeño que $p\_mem \cdot pop.tam$.

4.2.4 Algoritmo Memético Mejorado - AMM

Esta implementación del algoritmo memético es idéntica a la anterior salvo en el uso de *localSearch*. En este casi en lugar de aplicar la búsqueda local clásica utilizaremos la búsqueda local que mejores resultados dio en la práctica anterior: la búsqueda local determinista o *Deterministic Local Search* (LDS) explicada en profundidad la sección 4.1.3. Así mismo aplicaremos la misma factorización en la evaluación de las soluciones explicada en la sección anterior.

4.2.5 Búsqueda Local Determinista - LDS

De cara a comparar los resultados obtenidos en esta práctica se ha retocado la búsqueda local determinista de la práctica anterior para que la forma de medir el número máximo de evaluaciones sea la misma, implementando la factorización ya explicada. De esta forma fijamos el mismo número máximo de evaluaciones para que la comparación es consistente.

5 Procedimiento de desarrollo

5.1 Práctica 1

Todos los algoritmos se han implementado en $C++$ y se encuentran en la carpeta adjunta. El código esta dividido independientes y autosuficientes que contienen cada uno un algoritmos de los explicados anteriormente. Adicionalmente hay dos archivos más para el tratamiento de los datos.

Se han implementado también una serie de scripts en *bash*, así como un $makefile$ que permite la automatización de todo el proceso. El $makefile$ tiene esencialmente dos tipos de comandos: $examples < algoritmo >$ y $measure < algoritmo >$, donde $< algoritmo > \in Greedy, LS, LSD, GS$. El primer comando compila y ejecuta $< algoritmo >$ para tres ejemplos, uno de cada set de entrenamiento. El segundo comando ejecuta 200 veces $< algoritmo >$ en cada caso del problema y hace la media de los datos proporcionados para cada uno de los casos. Los datos de salida se almacenan en $output/ < algoritmo > .dat$.

Finalmente el comando *measureAll* ejecuta *measure* sobre todos los algoritmos excepto *greedy*, ya que no tienen ningún componente aleatorio. Para tomar datos sobre el algoritmo *greedy* ejecutaremos *measureGreedy* una única vez.

Para los experimentos 1 y 2 se han utilizado los comandos descritos anteriormente, con 200 repeticiones por algoritmo.

Para el experimento 3 se han alterado ligeramente los fuentes y se ha ejecutado de nuevo el comando *measureAll*, con 10 repeticiones por algoritmo. En este caso se ha calculado el máximo de los $fitness$ en vez de la media, manteniendo los cálculos para el tiempo y las iteraciones.

Para el experimento 4 se ha utilizado el script *evolution.sh* modificando ligeramente los fuentes para obtener los datos requeridos, así como LibreOffice para realizar las gráficas.

Todos los experimentos se han realizado en una misma máquina con las siguientes características: Sistema operativo Ubuntu 18.04.2, 7.7Gib de memoria y procesador IntelCore i7-7700@3.60GHzx8.

5.2 Práctica 2

Como es natural he reutilizado toda la automatización preparada en la práctica anterior, manteniendo principalmente los comandos $examples < algoritmo >$, $measure < algoritmo >$ y $measureAll$.

De la misma forma he lanzado todos los programas en la misma máquina que en la práctica anterior para que la comparación entre algoritmos tenga sentido.

En cuanto a la semilla para los generadores aleatorios, esta se inicializa de forma aleatoria para que las múltiples mediciones tomadas sean distintas.

Para el experimento uno se han utilizado los siguientes valores para los parámetros:

- Tamaño de la población: 50 cromosomas
- Máximas evauaciones totales: 50000
- Probabilidad de mutación: 0.001
- Probabilidad de cruce en AGE: 1
- Probabilidad de cruce en AGG y AM: 0.7
- En el tipo 1 de AM, valor de p_mem: 0.1
- Para AM, máximas iteraciones de la búsqueda local: 40

Para el experimento 1 se han ejecutado todos los algoritmos 30 veces, mientras que para los algoritmos que aparecen en el experimento 2 y no en el 1 (LSD y AMM) unicamente 2 veces por algoritmo, debido a la falta de tiempo.

Para el tercer experimento se utiliza el script *evolution*. Para ello basta con utilizar el comando *makeevolution*, que ejecuta los algoritmos $AGEu$, $AM1$ y AMM sobre los conjuntos de datos $MDG - a\_21$ y $SOM - b\_13$ midiendo la evolución en cada generación de la mejor solución de la población.

6 Experimentos realizados

6.1 Práctica 1

En esta sección describiremos los experimentos realizados y estudiáremos los resultados obtenidos.

6.1.1 Experimentos 1: resultados iniciales

En primer lugar he tomado datos como se expone en la sección anterior. Adjunto a continuación las tablas de los tiempos y los resultados obtenidos para los cuatro algoritmos mencionados.

	Algoritmo Greedy	Búsqueda Local	BS Determinista	BS y Greedy	Óptimos
MDG-a_1	112135	105832	106473	112139	114259
MDG-a_2	112109	105962	106416	112118	114327
MDG-a_3	112541	105776	106267	112541	114123
MDG-a_4	112590	105884	106318	112590	114040
MDG-a_5	112204	106118	106598	112255	114064
MDG-a_6	112347	106099	106568	112359	114204
MDG-a_7	112583	105940	106446	112590	114338
MDG-a_8	112023	105829	106320	112033	114158
MDG-a_9	112411	106290	106520	112429	114132
MDG-a_10	112634	105981	106536	112634	114197
SOM-b_11	20420	19181	20619.6	20432.8	20743
SOM-b_12	35574	33798.3	35689.9	35587.3	35881
SOM-b_13	4542	3955.94	4541.65	4544.6	4658
SOM-b_14	16888	15500.5	16878.1	16888	16956
SOM-b_15	36065	33870.9	36259.9	36065	36317
SOM-b_16	62294	59575.7	62722.7	62297	62487
SOM-b_17	6938	6135.27	6991.79	6938	7141
SOM-b_18	25843	23954.5	25826.1	25843	26258
SOM-b_19	55406	52741.8	55894.9	55413.4	56572
SOM-b_20	96593	92657.6	96398.7	96594	97344
GKD-c_21	17831.8	16483	18051.3	17833.3	19485,1875
GKD-c_22	17973	16797.3	18188.1	17973	19701,53711
GKD-c_23	17973.3	16574.8	18036.6	17973.3	19547,20703
GKD-c_24	17860	16384.7	18052.7	17866.3	19596,46875
GKD-c_25	17822.9	16702.6	18013.9	17822.9	19602,625
GKD-c_26	17741.1	16505.8	17991.6	17741.1	19421,55078
GKD-c_27	18016.5	16567.3	18161.9	18016.5	19534,30664
GKD-c_28	17897.7	16522.3	18089.3	17897.7	19487,32031
GKD-c_29	17784.5	16374.5	17817.6	17787.1	19221,63477
GKD-c_30	17860.1	16578	18042.7	17899.2	19703,35156

Table 2: Experimento 1 - Costes

	Greedy	Búsqueda local	BS determinista	BS greedy
MDG-a_1	0.249346	1.29837	1.67138	0.259307
MDG-a_2	0.251498	1.25412	1.64122	0.299649
MDG-a_3	0.25458	1.32799	1.64147	0.281972
MDG-a_4	0.264134	1.30341	1.66133	0.310694
MDG-a_5	0.258874	1.30089	1.71872	0.285821
MDG-a_6	0.259026	1.33751	1.74401	0.334614
MDG-a_7	0.260951	1.34526	1.69168	0.297083
MDG-a_8	0.258524	1.35116	1.71162	0.314464
MDG-a_9	0.257905	1.43386	1.76812	0.259414
MDG-a_10	0.252341	1.3025	1.74672	0.262792
SOM-b_11	0.003922	0.0769007	0.094339	0.0115729
SOM-b_12	0.005933	0.168331	0.178364	0.0135729
SOM-b_13	0.001846	0.0198992	0.0233178	0.00484763
SOM-b_14	0.004533	0.0882583	0.11522	0.010587
SOM-b_15	0.008492	0.189072	0.360455	0.0206853
SOM-b_16	0.013224	0.326374	0.634729	0.0254535
SOM-b_17	0.003302	0.0304316	0.0500629	0.00627894
SOM-b_18	0.008476	0.126879	0.306938	0.0167809
SOM-b_19	0.016259	0.260259	0.765596	0.071728
SOM-b_20	0.025305	0.456821	1.14616	0.0639032
GKD-c_21	0.003234	0.0399083	0.0699487	0.00735801
GKD-c_22	0.003143	0.0509766	0.0709061	0.00732955
GKD-c_23	0.003172	0.0460402	0.0729913	0.01287
GKD-c_24	0.003158	0.0489642	0.0817394	0.0074404
GKD-c_25	0.003188	0.0501347	0.0702928	0.00716391
GKD-c_26	0.003173	0.0519859	0.0584565	0.00720639
GKD-c_27	0.004301	0.0562678	0.0815131	0.00846629
GKD-c_28	0.003306	0.0476838	0.0648686	0.00651156
GKD-c_29	0.003221	0.0512277	0.0611578	0.00678151
GKD-c_30	0.00325	0.0478213	0.0706898	0.00764579

Table 3: Experimento 1 - Tiempos (s)

	Greedy	Búsqueda Local	BL Determinista	BL con greedy
Desv	3,81	10,25	5,12	3,79
Tiempo (s)	0,09	0,52	0,71	0,11

Table 4: Experimento 1 - Comparativa entre algoritmos

6.1.1.1 Análisis

Comencemos observando la tabla 4 para obtener una visión general de los resultados. Por un lado, los tiempos de las búsquedas locales clásica y determinista no son para nada sorprendentes: Ambas son notablemente más lentas que el algoritmo *Greedy*, siendo la BLD ligeramente más lenta que la BL. Sin embargo la media de tiempos para la $BLconGreedy$ es realmente sorprendente. Profundizaremos en este aspecto en el experimento 2.

Por otro lado el algoritmo *Greedy* obtiene un valor *Desv* menor que los de la búsquedas locales clásica y determinista. Sin embargo, si miramos la tabla 2 con más detenimiento nos

damos cuenta de que efectivamente sobre los casos MDG el algoritmo $Greedy$ obtiene mejores costes, pero en los demás casos la BLD suele superarlo. Esta es justamente la motivación de cara a la $BLcongreedy$.

Estos resultados son un claro ejemplo de como un algoritmo puede ser más efectivo frente a un caso de estudio particular y no serlo en otro. Así mismo nos damos cuenta de que la media en este caso no es tan representativa como nos gustaría. Aunque para esta práctica no lo he tenido en cuenta, para las próximas calcularé también la desviación típica para estos valores.

En cuanto a la $BLconGreedy$ era de esperar que tuviese valores como mínimo mejores que los del $Greedy$. Observamos que sin embargo estos valores no son ni mucho menos notablemente mejores. Surgen por lo tanto dos preguntas: Por qué el tiempo de ejecución de $BLconGreedy$ es tan sumamente bajo y por qué su coste apenas mejora respecto al $Greedy$. Ambas preguntas serán contestadas en el segundo experimento. La hipótesis es relativamente evidente: parece que la $BLcongreedy$ no explota el entorno tanto como nos gustaría.

6.1.2 Experimento 2: exploración del vecindario

Motivado por las anteriores cuestiones he estudiado las iteraciones que toma cada algoritmo de búsqueda local hasta converger. Esto nos permite estudiar por un lado si la BLD merece la pena y por otro comparar la convergencia de forma general. Estos han sido los resultados obtenidos.

	Búsqueda Local	BS Determinista	BS con greedy
MDG-a_1	333.34	458.93	1.05
MDG-a_2	341.42	460.72	2.25
MDG-a_3	333.32	445.39	0
MDG-a_4	334.15	450.24	0
MDG-a_5	341.59	466.07	3.9
MDG-a_6	350.235	484.405	1.5
MDG-a_7	342.575	452.6	2.25
MDG-a_8	337.055	477.755	2.3
MDG-a_9	357.475	479.48	3.85
MDG-a_10	341.795	472.395	0
SOM-b_11	77.785	371.73	1.15
SOM-b_12	102.11	466.15	2.3
SOM-b_13	44.015	189.585	1
SOM-b_14	97.2	378.75	0
SOM-b_15	115.06	543.42	0
SOM-b_16	143.41	674.185	1
SOM-b_17	56.64	252.86	0
SOM-b_18	119.47	511.075	0
SOM-b_19	145.545	720.03	1.5
SOM-b_20	179.53	785.815	1
GKD-c_21	38.005	255.305	1
GKD-c_22	48.95	281.135	0
GKD-c_23	42.24	268.165	0
GKD-c_24	37.6	257.63	1
GKD-c_25	50.375	281.54	0
GKD-c_26	41.32	258.45	0
GKD-c_27	38.74	306.75	0
GKD-c_28	42.035	259.42	0
GKD-c_29	41.83	281.97	1.3
GKD-c_30	42.225	278.765	2.05

Table 5: Experimento 2 - Estudio de convergencia (iteraciones)

6.1.2.1 Análisis

Observando los resultados quedan claramente resueltas las cuestiones planteadas inicialmente. La convergencia es terriblemente prematura en el caso de la $BLconGreedy$, y esto explica tanto los bajos valores de tiempos como la reducida mejora en coste respecto al *Greedy* inicial. Conjeturo que esta convergencia se debe a la cota en el número de vecinos generados impuesto (50.000), a la manera de explorar el vecindario y a la calidad de la solución inicial obtenida con el *Greedy*.

Por otra parte observamos que el número de iteraciones para la BLD es considerablemente mayor que para la BL, llegando hasta a quintuplicarse en ocasiones, como es el caso de algunos $GKD - c$. Esto no indica que los reajustes realizados realmente merecen la pena en cuanto a costes, si bien en tiempo puede que no sea el caso si ejecutamos esta búsqueda de forma reiterada.

En vista de estos últimos resultados otra alternativa interesante sería aplicar la BLD con solución inicial la obtenida por el *Greedy*. Esta búsqueda completamente determinista (BTD) sería un algoritmo 100% determinista que explotaría en profundidad el entorno de la solución inicial.

6.1.3 Experimento 3: búsqueda del óptimo

En el primer experimento podíamos observar como en algunos casos del problema obteníamos valores mejores que los óptimos conocidos, como es el caso de $SOMB - b_16$ en la tabla 2. Dicha observación propicia el siguiente experimento, en este caso buscando la mejor solución posible. Para ello se han utilizado las tres búsquedas ya explicadas pero eliminando la cota en la generación del vecindario. Ejecutamos cada búsqueda un total de 10 veces, tomando el máximo sobre los costes obtenidos y realizando la media para los tiempos y las iteraciones. Estos son los resultados obtenidos.

	Búsqueda Local	BS Determinista	BS y Greedy	Óptimos
MDG-a_1	113776	113308	113054	114259
MDG-a_2	113413	113288	113241	114327
MDG-a_3	113314	113171	113293	114123
MDG-a_4	113629	113300	113248	114040
MDG-a_5	113613	113112	113399	114064
MDG-a_6	113347	113368	113338	114204
MDG-a_7	113746	113457	113450	114338
MDG-a_8	113401	113343	113314	114158
MDG-a_9	113655	113383	113036	114132
MDG-a_10	113482	113488	113543	114197
SOM-b_11	20704	20585	20650	20743
SOM-b_12	35745	35646	35742	35881
SOM-b_13	4653	4608	4615	4658
SOM-b_14	17020	16924	16948	16956
SOM-b_15	36364	36319	36327	36317
SOM-b_16	62806	62767	62699	62487
SOM-b_17	7057	7028	6975	7141
SOM-b_18	25977	25901	25953	26258
SOM-b_19	56210	56056	55992	56572
SOM-b_20	97221	97197	97213	97344
GKD-c_21	18072.1	18040.1	18034.7	19485,1875
GKD-c_22	18193.7	18207.3	18211.3	19701,53711
GKD-c_23	18057.1	18034.4	18065.4	19547,20703
GKD-c_24	18060.5	18094.6	18101.7	19596,46875
GKD-c_25	18017.4	18021.5	18057.6	19602,625
GKD-c_26	18024.5	18021.8	17918.3	19421,55078
GKD-c_27	18163.7	18162.8	18158.6	19534,30664
GKD-c_28	18109.2	18109.2	18109.2	19487,32031
GKD-c_29	17837.2	17814.5	17831.3	19221,63477
GKD-c_30	18059.7	18046.9	18066.9	19703,35156

Table 6: Experimento 3 - Costes

	Búsqueda Local	BS Determinista	BS con greedy
MDG-a_1	172.293	40.2909	40.6578
MDG-a_2	160.427	41.9321	54.0231
MDG-a_3	166.142	47.3654	41.1937
MDG-a_4	171.914	39.4863	32.7488
MDG-a_5	174.249	40.8823	46.0528
MDG-a_6	158.605	43.6646	41.4238
MDG-a_7	171.656	44.7278	30.9779
MDG-a_8	177.631	42.4097	47.5606
MDG-a_9	162.381	45.3496	30.131
MDG-a_10	175.787	42.4186	52.1171
SOM-b_11	10.8909	0.080255	2.39299
SOM-b_12	15.4952	0.205982	2.52537
SOM-b_13	2.80544	0.0177933	0.398986
SOM-b_14	8.63016	0.112358	0.758402
SOM-b_15	19.0671	0.299473	4.09818
SOM-b_16	34.3534	0.616403	7.13911
SOM-b_17	4.20101	0.0535963	0.517853
SOM-b_18	14.6231	0.259467	1.38322
SOM-b_19	30.7497	0.930221	10.1462
SOM-b_20	57.6885	1.81487	12.5458
GKD-c_21	7.05474	0.0638733	1.89784
GKD-c_22	5.73836	0.059613	2.15262
GKD-c_23	6.41487	0.0687667	1.47689
GKD-c_24	8.16905	0.076535	3.02903
GKD-c_25	6.02595	0.095074	2.46421
GKD-c_26	5.69318	0.0487363	2.07736
GKD-c_27	7.73885	0.0706207	1.27569
GKD-c_28	5.64627	0.0579403	2.53345
GKD-c_29	5.48938	0.0627603	0.75972
GKD-c_30	6.84102	0.0698153	1.38801

Table 7: Experimento 3 - Tiempos(s)

	Búsqueda Local	BS Determinista	BS con greedy
MDG-a_1	1771.4	1542.67	207.4
MDG-a_2	1715.8	1548.33	260.4
MDG-a_3	1739.4	1649.67	173.4
MDG-a_4	1813.9	1621	150
MDG-a_5	1794.6	1568.33	217.4
MDG-a_6	1738.9	1598	222
MDG-a_7	1780.3	1656.33	154.8
MDG-a_8	1790.1	1505.33	219.2
MDG-a_9	1712.9	1667	155
MDG-a_10	1766.1	1559.33	230.8
SOM-b_11	423.6	316	67
SOM-b_12	511.2	457	44.6
SOM-b_13	198.2	179	15.4
SOM-b_14	418.3	359	13.4
SOM-b_15	600.8	494.667	68.2
SOM-b_16	784.5	707.333	91
SOM-b_17	277.8	246	12.6
SOM-b_18	566.1	476	27.4
SOM-b_19	819.7	759	155.6
SOM-b_20	1073.6	913	150.8
GKD-c_21	333.9	246	56.2
GKD-c_22	306.3	264.667	84.8
GKD-c_23	320.4	274	33.8
GKD-c_24	333.8	263.333	72.6
GKD-c_25	291.7	314	72.8
GKD-c_26	302.7	178	50.4
GKD-c_27	337.5	283	42.4
GKD-c_28	278.9	267	70.8
GKD-c_29	298	255	19.4
GKD-c_30	317.4	300.667	53.4

Table 8: Experimento 3 - Iteraciones

	Búsqueda Local	BL Determinista	BL con greedy
Desv	2,79	2,98	2,98
Tiempo (s)	65,15	14,45	15,93
Iteraciones	880,59	782,29	106,43

Table 9: Experimento 3 - Comparativa entre algoritmos

6.1.3.1 Análisis

Mirando a las tablas 4 y 9 vemos como los valores de *Desv* han bajado notablemente, especialmente para la *BL* y la *BLD*, mientras que los tiempos se han disparado. Aunque el *Desv* obtenido para la *BL* es menor que el de las otras dos búsquedas achaco esto al reducido número de iteraciones realizadas: unicamente 10 para cada algoritmo.

De nuevo nos fijamos en la tabla 7 para comprobar los resultados caso a caso. Si bien los mejores valores conocidos unicamente se sobrepasan para las entradas $SOM - b_1 4$, $SOM - b_1 5$

y $SOM-b_16$ los tres algoritmos se acercan notablemente a dichos valores para el resto de casos.

A pesar de los que los costes obtenidos en los distintos algoritmos son sumamente parecidos, no es ni mucho menos el caso para los tiempos y las iteraciones. Se comprueba sin duda la efectividad de la exploración del entorno implementada para la BLD observando los estos valores. Además, se comprueba la hipótesis respecto a la pobre ejecución de la $BLconGreedy$ cuando limitamos la exploración. Esto refuerza la idea de la BTD mencionada en el experimento 1.

Finalmente observamos en la iteraciones obtenidas como la convergencia de la BLD es superior a la de la BL. Obviamente la de la $BLconGreedy$ es muy superior puesto que comienza mucho más cerca del óptimo. Estos razonamientos propician un último experimento de cara a estudiar la convergencia de las soluciones. Si bien me habría gustado realizarlo sin cota máxima como en este último experimento, no me ha sido posible por cuestiones de tiempo. Por ende, considerar el algoritmo $BLconGreedy$ en el siguiente experimento tampoco tiene sentido, ya que la convergencia es extremadamente prematura.

6.1.4 Experimento 4: estudio evolutivo de la exploración del entorno

Estudiamos la convergencia de la búsqueda local clásica y la determinista iteración a iteración. Para ello ejecutamos estos algoritmos sobre los conjuntos de datos $MDG-a_21$ y $SOM-b_13$, escogidos por tener comportamientos particularmente distintos en las tablas 2 y 5. Volvemos a utilizar la cota en la exploración del vecindario, y en cada iteración obtenemos el coste actual de la solución en estudio. Podemos ver los resultados obtenidos en las siguientes gráficas.

6.1.4.1 Análisis

Podemos ver como en los distintos casos la evolución es particularmente distinta. Por un lado, para $MDG-a_21$ la convergencia de ambos algoritmos es asintóticamente idéntica, manteniéndose la BLD por encima en parte debido a la solución inicial ligeramente mejor.Por otro, en $SOM-b_13$ la mejora de la BL respecto a la BLD es sustancial pero esta se estanca rápidamente debido a que su exploración del espacio es peor. Este es otro ejemplo de como los mismos algoritmos pueden comportarse de formas completamente distintas para entradas diferentes.

6.2 Práctica 2

6.2.1 Experimento 1: resultados iniciales

Para este primer experimento se han estudiado los dos AGG, los dos AGE y los tres AM definidos anteriormente. Utilizando los parámetros definidos en 5.2, lanzamos los algoritmos 30 veces por test-case cada uno y tomamos datos del tiempo, fitness y generaciones realizadas por cada uno. Presentamos a continuación las medias de estos datos:

	AGGu	AGGp	AGEu	AGEp	AM0	AM1	AM1
MDG-a_1	108891	103366	109277	104244	110992	111932	111256
MDG-a_2	108830	103784	109242	104577	111723	111468	111536
MDG-a_3	108791	103647	109261	103492	111620	111296	111484
MDG-a_4	108841	103714	109291	103558	111886	111738	111445
MDG-a_5	108549	104275	109292	104132	111356	111566	111216
MDG-a_6	108694	104393	109093	104210	111372	111768	111441
MDG-a_7	108899	103750	109418	104317	111176	111125	111759
MDG-a_8	108681	103860	109045	104168	111723	111738	111092
MDG-a_9	108659	104258	109435	103738	111401	111734	111257
MDG-a_10	108775	102935	109155	104006	111837	110936	111396
SOM-b_11	20206.7	19432	20268	19401	20409.5	20371	20333.5
SOM-b_12	35246.8	34127.5	35340.9	34216	35432	35379.5	35372.5
SOM-b_13	4333.85	4049.5	4368.15	4148	4435	4484	4444
SOM-b_14	16305.5	15547.5	16424.2	15521.5	16623	16477	16582.5
SOM-b_15	35615.8	34058	35647.8	34386	35622.5	35846	35817.5
SOM-b_16	61871.5	59912.5	61995.1	60079.5	62317	62406.5	62044
SOM-b_17	6605.75	6290.5	6694.45	6289	6876.5	6904	6749.5
SOM-b_18	25023.5	23966	25206	23889.5	25259	25481	25396
SOM-b_19	54865.7	53187	55060.9	52865.5	55603	55468	55556
SOM-b_20	95792.9	92998.5	96038.6	93193	96319.5	96633.5	96542.5
GKD-c_21	17608.8	17017	17645.8	17128.3	17915.9	17899.7	17918.7
GKD-c_22	17756.5	17155.2	17834	17138.8	18112.5	17973.1	17888.4
GKD-c_23	17718	16929	17732.2	17027.1	17857.5	17968.2	17932.2
GKD-c_24	17749.5	16981.2	17746.4	17104.2	17791.6	17749.5	17963.5
GKD-c_25	17658.9	16816.9	17697.5	16992.3	17864.3	17820	17833.4
GKD-c_26	17582.4	17116.2	17615.8	17145.5	17856.2	17790.8	17843.8
GKD-c_27	17810	17096.8	17850.3	17070.8	17977.6	18038.6	17998.8
GKD-c_28	17649.3	17063	17734.5	17089.3	17948.9	18002.1	17819.3
GKD-c_29	17414.2	16761	17454.3	16822.8	17639.5	17532.5	17543.2
GKD-c_30	17721.8	16915.6	17706.5	17085.6	17850.2	17969.3	17963.6

Table 10: Experimento 1 - Costes

	AGGu	AGGp	AGEu	AGEp	AM0	AM1	AM1
MDG-a_1	24.4233	23.8637	24.1219	23.6219	21.9402	23.2932	22.797
MDG-a_2	24.405	23.8894	24.2071	23.4312	22.0997	22.2993	23.3811
MDG-a_3	24.3328	23.852	24.3339	23.8606	23.2322	23.1624	23.1435
MDG-a_4	24.3705	23.9463	24.1634	23.3537	22.8015	23.107	23.0078
MDG-a_5	24.4211	23.7319	24.2868	23.9673	21.8929	22.533	22.8683
MDG-a_6	24.4364	24.5861	24.3023	23.8618	22.3339	22.3768	23.0249
MDG-a_7	24.3507	24.0536	24.1702	23.6154	22.6294	22.3369	23.1606
MDG-a_8	24.4853	23.6647	24.0161	23.2876	22.6126	22.6438	22.8375
MDG-a_9	24.4153	23.7216	24.2415	24.2212	22.7098	22.8243	21.8571
MDG-a_10	24.3466	23.4462	24.1385	24.1119	22.6193	22.8244	22.7726
SOM-b_11	1.20194	1.18055	1.20846	1.1629	1.24611	1.20307	1.20698
SOM-b_12	1.59528	1.53491	1.57967	1.58228	1.94155	1.61071	1.58557
SOM-b_13	0.868631	0.812625	0.857292	0.891678	0.438967	0.734924	0.853313
SOM-b_14	1.5129	1.48241	1.47245	1.42183	1.22705	1.41789	1.48675
SOM-b_15	2.09646	2.00823	2.08278	2.19775	2.40745	2.12159	2.08059
SOM-b_16	2.76473	2.65904	2.72259	2.68241	3.72652	2.86986	2.78566
SOM-b_17	1.37386	1.30215	1.3465	1.32537	0.740074	1.19312	1.35607
SOM-b_18	2.35214	2.22628	2.29393	2.21181	2.1313	2.30068	2.35398
SOM-b_19	3.30494	3.23536	3.29586	3.2272	4.0674	3.36809	3.38204
SOM-b_20	4.31045	4.23414	4.30605	4.14887	5.91416	4.61186	4.42713
GKD-c_21	1.36727	1.32412	1.35354	1.25208	0.765945	1.2595	1.33338
GKD-c_22	1.3738	1.30634	1.35173	1.24574	0.728988	1.20548	1.31984
GKD-c_23	1.37431	1.28311	1.35425	1.33676	0.767462	1.25363	1.33645
GKD-c_24	1.35535	1.3561	1.33929	1.31544	0.751525	1.16892	1.34858
GKD-c_25	1.36712	1.27663	1.33415	1.32144	0.733905	1.225	1.34138
GKD-c_26	1.37671	1.34339	1.35847	1.33036	0.740275	1.25453	1.38125
GKD-c_27	1.39016	1.31565	1.34685	1.30794	0.768023	1.18297	1.41582
GKD-c_28	1.37054	1.34797	1.33595	1.27007	0.732937	1.20843	1.31878
GKD-c_29	1.34627	1.30072	1.36707	1.29795	0.754039	1.23594	1.4178
GKD-c_30	1.33805	1.30159	1.34684	1.32056	0.751667	1.19237	1.31282

Table 11: Experimento 1 - Tiempos (s)

	AGGu	AGGp	AGEu	AGEp	AM0	AM1	AM1
MDG-a_1	1997.5	1995.5	24975	24975	1320.5	1904.5	1980
MDG-a_2	1996.35	1997	24975	24975	1368.5	1906	1976.5
MDG-a_3	1997.75	1997	24975	24975	1357	1904	1976.5
MDG-a_4	1995.6	1996	24975	24975	1370.5	1903	1974.5
MDG-a_5	1997.3	1995.5	24975	24975	1328	1908.5	1976.5
MDG-a_6	1996.3	1996	24975	24975	1340	1897	1969
MDG-a_7	1997.9	1998	24975	24975	1328	1903	1978
MDG-a_8	1997.6	1999.5	24975	24975	1334	1902	1979
MDG-a_9	1997	1992.5	24975	24975	1369	1903.5	1977.5
MDG-a_10	1997.8	2001	24975	24975	1348	1902.5	1975.5
SOM-b_11	1997.55	1992.5	24975	24975	1020	1826.5	1955.5
SOM-b_12	1996.75	1994	24975	24975	1139.5	1866.5	1963.5
SOM-b_13	1995.8	1995.5	24975	24975	591	1624	1910
SOM-b_14	1997.45	1994	24975	24975	873.5	1794.5	1949
SOM-b_15	1995.45	1997	24975	24975	1115	1856	1965.5
SOM-b_16	1997.45	1998	24975	24975	1340	1896.5	1980
SOM-b_17	1996.2	1993.5	24975	24975	668	1696.5	1927
SOM-b_18	1995.5	1995	24975	24975	1025	1824.5	1958.5
SOM-b_19	1997.1	1994	24975	24975	1310	1894	1974.5
SOM-b_20	1997.55	1997	24975	24975	1372	1908.5	1971.5
GKD-c_21	1997.3	1995.5	24975	24975	704.5	1699.5	1929.5
GKD-c_22	1996.6	2000	24975	24975	675	1694	1922
GKD-c_23	1995	1995	24975	24975	705	1689	1928.5
GKD-c_24	1997.1	1996.5	24975	24975	720	1706	1928
GKD-c_25	1995.6	1997.5	24975	24975	680	1692.5	1925
GKD-c_26	1997.1	1992.5	24975	24975	679.5	1693	1918.5
GKD-c_27	1994.9	1997	24975	24975	699	1688	1925.5
GKD-c_28	1998.4	1997.5	24975	24975	670	1685.5	1917
GKD-c_29	1996.05	1995.5	24975	24975	681.5	1695.5	1922
GKD-c_30	1996.65	1994	24975	24975	700.5	1698.5	1924

Table 12: Experimento 1 - Generaciones

	AGGu	AGGp	AGEu	AGEp	AM0	AM1	AM2
Desv	5,93	9,84	5,56	9,56	4,33	4,27	4,44
Tiempo (s)	9,30	9,09	9,22	9,04	8,54	8,70	8,80
Generaciones	1996,75	1995,98	24975,00	24975,00	1027,75	1805,43	1951,93

Table 13: Experimento 1 - Comparativa entre algoritmos

6.2.1.1 Análisis

Comencemos observando la tabla 13 para obtener una **visión general** de los resultados obtenidos. Podemos apreciar consultando la fila *Desv* que los algoritmos que mejores resultados obtienen son siempre los meméticos. En cuanto a los genéticos, tanto en la estrategia generacional como en la estacionaria el resultado es claro: el operador de cruce uniforme es notablemente mejor que el posicional (al menos combinándolo con el resto de operadores implementados y para los parámetros utilizados). Es por ello que en futuros experimentos no estudiaremos ni AGGp ni AGEp.

En cuanto a los **tiempos**, los resultados son realmente consistentes. Si bien los meméticos son un poco más rápidos, la diferencia no llegar a ser notable. Consultando la tabla 11 vemos que para las entradas más grandes (MDG), los valores de todos los algoritmos oscilan entre 21 y 24 segundos. Ante tan breve variación podemos asumir que el "mejor" algoritmo será el que mejor valores obtenga, considerando que hemos fijado el número máximo de iteraciones.

Volvemos a 13 para observar el número medio de **generaciones** obtenido. En primer lugar puede parecer curioso que ambos algoritmos AGE obtengan exactamente el mismo valor medio. Adicionalmente, si observamos la tabla 12 podemos ver como se obtienen 24975 generaciones para todos los experimentos (recordemos que este valor a su vez es una media entre 30 ejecuciones). Sin embargo, no podía ser de otra forma: el número de evaluaciones por generación de este algoritmos esta fijado de antemano, es exactamente dos. Considerando las 50 evaluaciones iniciales para inicializar la población, es lo esperado obtener exactamente $(50.000 - 50)/2 = 24.975$ generaciones.

Por otro lado, realizando una operación sencilla ($n_evaluaciones/n_generaciones$) vemos que el número de evaluaciones por generación obtenido para los AGG oscila entorno a $2,5$. Los algoritmos meméticos obtienen valores parecidos exceptuando el $AM0$, que ejecuta la búsqueda local para toda la población cada 10 iteraciones, obtiene 5 una media aproximada de 5 evaluaciones por generación.

Finalmente comparamos la eficacia de los algoritmos AGGu y AGEu. Para ello observamos la fila Desv en 13 y los test-cases MDG en la tabla 10. En ellos podemos ver como el AGEu obtiene valores ligeramente mayores que el AGGu en la mayoria de los casos. Sin embargo estos casos se encuentran realmente alejados de los mejores valores que oscilan entre 113.000 y 114.000, como podemos ver en la columna de óptimos de la tabla 2. Como cabría esperar, todos los algoritmos meméticos obtienen mayores valores de $fitness$, siendo el mejor de ellos $AM1$, que aplicaba la búsqueda local a elementos arbitrarios de la población.

De cara al siguiente experimento restringimos el conjunto de algoritmos estudiado al $AGEu$ y $AM1$, añadiendo la búsqueda local de la práctica anterior y el algoritmo memético mejorado.

6.2.2 Experimento 2: Comparativa con práctica 1 y memético mejorado

En el experimento anterior utilizábamos unicamente los algoritmos obligatorios realizados en esta práctica para la comparación, de cara a evitar trabajar simultáneamente con nueve algoritmos distintos. Procedemos ahora a comparar los dos algoritmos que mejores resultados obtuvieron en el experimento anterior ($AGEu$ y $AM1$) con la búsqueda local que mejores resultados dio en la práctica anterior (LSD), junto con el algoritmo memético mejorado implementado para esta práctica.

Para los algoritmos $AGEu$ y $AM1$ utilizamos los datos obtenidos en el experimento anterior, mientras que para la LSD no tomaremos muestras de las evaluaciones realizadas ya que no tiene sentido compararlas con las generaciones de un algoritmo genético.

	AGEu	AM1	AMM	LSD
MDG-a_1	109277	111932	111603	105344
MDG-a_2	109242	111468	111771	106374
MDG-a_3	109261	111296	111254	106137
MDG-a_4	109291	111738	111930	106058
MDG-a_5	109292	111566	111902	106222
MDG-a_6	109093	111768	111227	106156
MDG-a_7	109418	111125	111918	106126
MDG-a_8	109045	111738	111388	105580
MDG-a_9	109435	111734	111247	106000
MDG-a_10	109155	110936	111274	105870
SOM-b_11	20268	20371	20595	20615.5
SOM-b_12	35340.9	35379.5	35698.5	35745
SOM-b_13	4368.15	4484	4556	4610
SOM-b_14	16424.2	16477	16882.5	16881
SOM-b_15	35647.8	35846	36100.5	36130
SOM-b_16	61995.1	62406.5	62670.5	62332.5
SOM-b_17	6694.45	6904	6914	6992
SOM-b_18	25206	25481	25764	25396
SOM-b_19	55060.9	55468	55845	55189
SOM-b_20	96038.6	96633.5	96832.5	95558
GKD-c_21	17645.8	17899.7	18048.3	18046
GKD-c_22	17834	17973.1	18206.8	18208.6
GKD-c_23	17732.2	17968.2	18001.7	18032.2
GKD-c_24	17746.4	17749.5	18051	18035.9
GKD-c_25	17697.5	17820	18001.9	18041
GKD-c_26	17615.8	17790.8	17945.6	17977.3
GKD-c_27	17850.3	18038.6	18121.5	18164.7
GKD-c_28	17734.5	18002.1	18085.8	18109.2
GKD-c_29	17454.3	17532.5	17809.5	17825.9
GKD-c_30	17706.5	17969.3	18045.5	18030.4

Table 14: Experimento 2 - Costes

	AGEu	AM1	AMM	LSD
MDG-a_1	24.1219	23.2932	12.9742	1.36342
MDG-a_2	24.2071	22.2993	13.3091	1.6805
MDG-a_3	24.3339	23.1624	13.1407	1.6231
MDG-a_4	24.1634	23.107	13.5073	1.47381
MDG-a_5	24.2868	22.533	13.3399	1.3582
MDG-a_6	24.3023	22.3768	13.4465	1.34014
MDG-a_7	24.1702	22.3369	13.1167	1.37532
MDG-a_8	24.0161	22.6438	13.3501	1.37907
MDG-a_9	24.2415	22.8243	13.5913	1.4398
MDG-a_10	24.1385	22.8244	13.3084	1.15298
SOM-b_11	1.20846	1.20307	0.744437	0.108794
SOM-b_12	1.57967	1.61071	1.15973	0.207137
SOM-b_13	0.857292	0.734924	0.379541	0.0220685
SOM-b_14	1.47245	1.41789	0.791695	0.120156
SOM-b_15	2.08278	2.12159	1.31182	0.31353
SOM-b_16	2.72259	2.86986	1.9723	0.515377
SOM-b_17	1.3465	1.19312	0.585186	0.0500965
SOM-b_18	2.29393	2.30068	1.23769	0.201794
SOM-b_19	3.29586	3.36809	2.07114	0.40222
SOM-b_20	4.30605	4.61186	3.05897	0.619741
GKD-c_21	1.35354	1.2595	0.594739	0.0729705
GKD-c_22	1.35173	1.20548	0.594813	0.0750675
GKD-c_23	1.35425	1.25363	0.574288	0.0886425
GKD-c_24	1.33929	1.16892	0.593927	0.079868
GKD-c_25	1.33415	1.225	0.590534	0.0821845
GKD-c_26	1.35847	1.25453	0.602524	0.0644635
GKD-c_27	1.34685	1.18297	0.582689	0.0856165
GKD-c_28	1.33595	1.20843	0.600887	0.0891885
GKD-c_29	1.36707	1.23594	0.590709	0.06394
GKD-c_30	1.34684	1.19237	0.593216	0.091104

Table 15: Experimento 2 - Tiempos (s)

	AGEu	AM1	AMM
MDG-a_1	24975	1904.5	770
MDG-a_2	24975	1906	770
MDG-a_3	24975	1904	770.5
MDG-a_4	24975	1903	770
MDG-a_5	24975	1908.5	770
MDG-a_6	24975	1897	770
MDG-a_7	24975	1903	770
MDG-a_8	24975	1902	770
MDG-a_9	24975	1903.5	770
MDG-a_10	24975	1902.5	770
SOM-b_11	24975	1826.5	810
SOM-b_12	24975	1866.5	945
SOM-b_13	24975	1624	795
SOM-b_14	24975	1794.5	824.5
SOM-b_15	24975	1856	810
SOM-b_16	24975	1896.5	780
SOM-b_17	24975	1696.5	776
SOM-b_18	24975	1824.5	770
SOM-b_19	24975	1894	775.5
SOM-b_20	24975	1908.5	770
GKD-c_21	24975	1699.5	775.5
GKD-c_22	24975	1694	772
GKD-c_23	24975	1689	777
GKD-c_24	24975	1706	776.5
GKD-c_25	24975	1692.5	771
GKD-c_26	24975	1693	771
GKD-c_27	24975	1688	778
GKD-c_28	24975	1685.5	776
GKD-c_29	24975	1695.5	772.5
GKD-c_30	24975	1698.5	775

Table 16: Experimento 2 - Generaciones

	AGEu	AM1	AMM	LSD
Desv	5,56	4,27	3,69	5,35
Tiempo (s)	9,22	8,70	5,08	0,58
Generaciones	24975,00	1805,43	783,37	-

Table 17: Experimento 2 - Comparativa entre algoritmos

6.2.2.1 Análisis

Comparando en primer lugar la LSD con el resto de algoritmos, vemos en la tabla 17 como obtenemos resultados ligeramente mejores que el $AGEu$, pero peores que el $AM1$. Sin embargo fijándonos en 14 caso a caso vemos como para las entradas de tipo MDG la búsqueda local obtiene valores considerablemente peores, mientras que en casos más reducidos queda por encima que el $AGEu$, como son las mayoría de los SOM y GKD. Adicionalmente, los tiempos obtenidos para LSD son ridículamente menores que los del resto de algoritmos.

En resumen podríamos deducir que para problemas de tamaño reducido la búsqueda local obtiene no solo mejores resultados sino tiempos muchos menores que los algoritmos genéticos, mientras que para casos mayores tienen un rendimiento bastante reducido.

Por otro lado, con el algoritmo memético mejorado observamos en 17 que se obtiene un mejor valor de $Desv$, si bien mirando caso a caso en la tabla 14 las soluciones no son notablemente mejores. Sin embargo si que se ha producido una mejora notable en cuanto a tiempos que se reducen casi un 50% en los mayores casos.

De igual forma que ocurría en el experimento 3 de la práctica anterior (6.1.3), el cambio en la búsqueda propicia una convergencia notablemente más rápida, si bien los valores alcanzados no presentan grandes diferencias. De igual forma que hizo el experimento 3 en la práctica anterior, estos resultados propician otro experimento para estudiar la evolución de las soluciones a nivel generacional.

6.2.3 Experimento 3: Estudio evolutivo a nivel generacional

Para este último experimento nos ceñimos a los algoritmos $AGEu$, $AM1$ y AMM. Así mismo utilizaremos unicamente dos conjuntos de datos, $MDG - a_2 1$ y $SOM - b_1 3$, por sus distintos tamaños. Realizamos una única ejecución para cada algoritmo por conjunto de datos. Presentamos dos gráficas distintas para cada conjunto de datos debido a la abismal diferencia en el número de generaciones.

6.2.3.1 Análisis

Podemos observar claramente en ambos conjuntos de datos como la hipótesis formulada en el experimento anterior era parcialmente errónea: el algoritmo AMM no converge más deprisa generacionalmente que el AM sino que ambos lo hacen extremadamente rápido, de igual forma que el $AGEu$. Observar este hecho nos permitiría dirigir nuestros esfuerzos a diseñar y utilizar otros operadores para explorar más el espacio de soluciones y no estancarnos tan rápidamente.

Aún así queda relativamente claro que quizás una estrategia evolutiva para este tipo de casos de prueba no sea el mejor procedimiento, ya que en general obtenemos tiempos mucho mayores que los de las búsquedas implementadas en la práctica anterior y si permitimos que las búsquedas exploren lo suficiente obtendremos resultados mejores y seguramente en un tiempo menor. Del mismo modo, para otros conjuntos de datos mayores que los utilizados las estrategias evolutivas podrían dar mejores resultados.