# Problema de Máxima Diversidad (MDP)

Técnicas basadas en poblaciones

Metaheurísticas: Práctica 2, Grupo 1

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

5 de mayo de 2019

# Índice

# 1. Introducción: práctica 2

De cara a la segunda práctica se han añadido a la memoria las secciones 2.2, 3.X, 4.2 y 5.2. 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.

**Algorithm 1:** singleContribution

```
Data: solution : sol , i : int
Result: contribution : double
begin
    contribution ← 0
    foreach j ∈ sol.v do
        contribution ← contribution + MAT[ i ][ j ]
    end
end
```

**Algorithm 2:** evaluateSolution

```
Data: solution : sol
Result: fitness : double
begin
    fitness ← 0
    foreach i ∈ sol.v do
        fitness ← fitness + SingleContribution ( sol, i )
    end
    fitness ← fitness / 2
end
```

## 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*.

**Algorithm 3:** evaluatesolution

**Data:** v : $vector < bool >$, elem: int
**Result:** result : double
**begin**
  result ← 0
  **foreach** $i \in [0, v.size)$ **do**
    result ← v[ i ] * MAT[ i ][ elem ]
  **end**
**end**

**Algorithm 4:** evaluatesolution

```
Data: sol : solution
Result: fitness : double
begin
    sol.fitness ← 0
    foreach i ∈ [0, sol.v.size) do
        if sol.v[ i ] then
            sol.fitness ← SingleContribution ( sol , i )
        end
    end
    // Counting twice all the possible distances
    sol.fitness ← sol.fitness / 2
    sol.evaluated ← true
end
```

### 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 $[a, b)$.

**Algorithm 5:** binaryTournament

```
Data: pop : population
Result: elem : int
begin
    r1 ← random(0, pop.tam)
    r2 ← random(0, pop.tam)
    if pop.v[r1].fitness > pop.v[r2].fitness then
        elem ← r1
    else
        elem ← r2
    end
end
```

Para el operador de mutación simplemente se cambia uno de los valores elegidos por otro

que no este escogido ya, de forma aleatoria:

**Algorithm 6:** mutateSolution

```
Data: pop : population
Result: elem : int
begin
    r1 ← random(0, pop.tam)
    r2 ← random(0, pop.tam)
    do
        r_on ← random(0, sol.v.size())
    while !sol.v[r_on]
    do
        r_off ← random(0, sol.v.size())
    while !sol.v[r_off] OR r_on == r_off
    sol.v[r_on] ← false
    sol.v[r_off] ← true
    if sol.evaluated then
        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
    end
end
```

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.

**Algorithm 7:** uniformCross

```
Data: p1 : solution, p2 : solution
Result: child : solution
begin
    child ← p1
    child.evaluated ← false
    m ← 0
    foreach i ∈ [0, p1.v.size()) do
        if p1.v[ i ] then
            m ← m + 1
        end
        if p1.v[ i ] AND p2.v[ i ] then
            child.v[ i ] ← true
        else if !p1.v[ i ] AND !p2.v[ i ] then
            child.v[ i ] ← false
        else
            child.v[ i ] ← random(0,2) == 0
        end
    end
    child ← repairSolution(child, m)
end
```

**Algorithm 8:** repairSolution

```
Data: child : solution, m : int
Result: child : solution
begin
    nTrues ← countTrues(child.v)
    while nTrues ¡m do
        r ← random(0, sol.v.size())
        if sol.v[ r ] then
            sol.v[r] ← false nTrues ← nTrues + 1
        end
    end
    while nTrues ¿m do
        r ← random(0, sol.v.size())
        if !sol.v[ r ] then
            sol.v[r] ← true nTrues ← nTrues - 1
        end
    end
end
```

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.

**Algorithm 9:** positionalCross

```
Data: p1 : solution, p2 : solution
Result: c1 : solution, c2 : solution
begin
    c1 ← p1
    c2 ← p2
    c1.evaluated ← false
    c2.evaluated ← false
    shuffled = vector¡int¿(size = c1.v.size(), value=false)
    foreach i ∈ [0, p1.v.size()) do
        if p1.v[ i ] AND p2.v[ i ] then
            c1.v[ i ] ← true
            c2.v[ i ] ← true
        else
            shuffled[ i ] ← true
            to_shuffle[ i ] ← p1.v[i]
        end
    end
    to_shuffle1 ← randomShuffle( to_shuffle )
    to_shuffle2 ← randomShuffle( to_shuffle )
    foreach i ∈ [0, p1.v.size()) do
        if shuffled[ i ] then
            c1.v[ i ] ← to_shuffle1[ i ]
            c2.v[ i ] ← to_shuffle2[ i ]
        end
    end
end
```

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.

**Algorithm 10:** replace

```
Data: pop : population, child1 : solution, child2 : solution
Result: pop : population
begin
    pop.v.append( child1 )
    pop.v.append( child2 )
    pop.v ← sort( pop.v )
    pop.v ← pop.v.resize( pop.v.size() - 2)
end
```

# 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.

**Algorithm 11:** farthestToSel

```
Data: selected : set < int > , non_selected : set < int >
Result: farthest : i
begin
    fitness ← 0
    max_sum_dist ← 0
    foreach i ∈ non_selected do
        current_sum_dist ← SingleContribution ( selected , i )
        if current_sum_dist > max_sum_dist then
            max_sum_dist ← current_sum_dist
            farthest ← i
        end
    end
end
```

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: $\{0, 1, \ldots, n\}$.

**Algorithm 12:** farthestToAll

```
Data: none
Result: farthest : i
begin
    all_elements ← {0, 1, ..., n}
    farthest ← farthestToSel( all_elements, all_elements )
end
```

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

**Algorithm 13:** greedy

```
Data: none
Result: selected : set < int >
begin
    non_selected ← {0, 1, ..., n}
    selected ← { farthestToAll() }
    while |selected| < m do
        farthest ← farthestToSel(selected, non_selected)
        non_selected ← non_selected ⋃ {farthest}
        selected ← selected - {farthest}
    end
end
```

### 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.

**Algorithm 14:** localSearch

```
Data: none
Result: sol : solution
begin
    sol ← randomSolution()
    stop ← false
    while !stop do
        stop , sol ← stepInNeighbourhood(sol)
    end
end
```

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.

**Algorithm 15:** operator<

```
Data: p1 : pair < int, double > , p2 : pair < int, double >
Result: comp : bool
begin
    comp ← p1.second < p2.second
end
```

A continuación reordenamos los elementos de la solución.

**Algorithm 16:** orderSolutionByContribution

```
Data: sol : solution
Result: sol : solution
begin
    pairs : vector < pair < int, double >>
    foreach i ∈ sol.v do
        pairs[ i ].first ← i
        pairs[ i ].second ← singleContribution(sol.v, i);
    end
    pairs ← sort(pairs, operator <)
    foreach i ∈ sol.v do
        sol.v[ i ] ← pairs[ i ].first
    end
end
```

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.

**Algorithm 17:** stepInNeighbourhood

```
Data: sol : solution
Result: stop : bool , sol : solution
begin
    sol ← orderSolutionByContribution(sol)
    max_i ← percentage_studied · |sol|
    max_randoms ← MAX/max_i
    stop ← true
    tries ← 0
    i ← 0
    while i < max_i do
        element_out ← sol.v[ i ]
        oldContribution ← singleContribution(sol.v, element_out)
        j ← rand(0, m)
        k ← 0
        while k < max_k do
            if j ∉ sol.v then
                newContribution ← singleContribution(sol.v, j) − MAT[j][element_out]
                if newContribution > oldContribution then
                    sol.v[ i ] ← j
                    sol.fitness ← sol.fitness + newContribution − oldContribution
                    pairs[ i ].first ← i
                    sol ← false
                    return
                end
                k ← k + 1
            end
            j ← rand(0, m)
        end
        i ← k + 1
    end
end
```

### 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*.

**Algorithm 18:** obtainBestOrdering

```
Data: sol : solution
Result: best_ordering : vector¡int¿
begin
    pairs : vector < pair < int, double >>
    foreach i ∈ 0, ..., n do
        if i ∉ sol.v then
            pairs[ i ].first ← i
            pairs[ i ].second ← singleContribution(sol.v, i);
        end
    end
    pairs ← sort(pairs, operator <)
    for i from |pairs| − 1 to 0 do
        best_ordering[ i ] ← pairs[ i ].first
    end
end
```

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, $localSearchDet$.

**Algorithm 19:** stepInNeighbourhoodDet

```
Data: sol : solution
Result: stop : bool , sol : solution
begin
    sol ← orderSolutionByContribution(sol)
    best_ordering ← obtainBestOrdering(sol)
    percent_i ← 0,1 max_i ← percent_i · |sol|
    max_k ← min(MAX/max_i, |best_ordering|)
    if max_k == |best_ordering| then
        max_i ← min(MAX/max_k, |sol|)
    end
    stop ← true
    i ← 0
    while i < max_i do
        element_out ← sol.v[ i ]
        oldContribution ← singleContribution(sol.v, element_out)
        k ← 0
        while k < max_k do
            j ← best_ordering[ k ] if j ∉ sol.v then
                newContribution ← singleContribution(sol.v, j) − MAT[j][element_out]
                if newContribution > oldContribution then
                    sol.v[ i ] ← j
                    sol.fitness ← sol.fitness + newContribution − oldContribution
                    pairs[ i ].first ← i
                    sol ← false
                    return
                end
                k ← k + 1
            end
        end
        i ← k + 1
    end
end
```

**Algorithm 20:** localSearchDet

```
Data: none
Result: sol : solution
begin
    sol ← randomSolution()
    stop ← false
    while !stop do
        stop , sol ← stepInNeighbourhoodDet(sol)
    end
end
```

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.

**Algorithm 21:** localSearchGreedy

```
Data: none
Result: sol : solution
begin
    sol ← greedy()
    stop ← false
    while !stop do
        stop , sol ← stepInNeighbourhoodDet(sol)
    end
end
```

### 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 | |

Tabla 1: Selección de operador por algoritmo

#### 4.2.1. Algoritmos Genéticos Generacionales - AGG

## 5. Procedimiento de desarrollo

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.

# 6. Experimentos realizados

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

## 6.1. 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 |

Tabla 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 |

Tabla 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 |

Tabla 4: Experimento 1 - Comparativa entre algoritmos

#### 6.1.1. 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.

## 6.2. 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 |

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

#### 6.2.1. 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.

## 6.3. 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 |

Tabla 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 |

Tabla 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 |

Tabla 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 |

Tabla 9: Experimento 3 - Comparativa entre algoritmos

### 6.3.1. 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 $SOM - b_14$, $SOM - b_15$

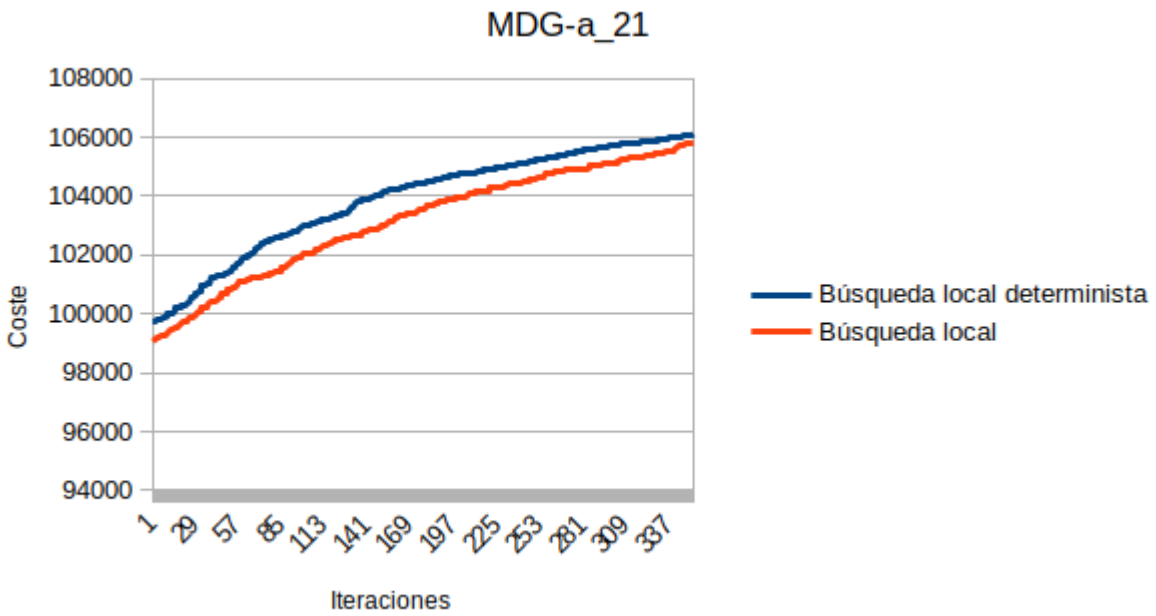
y $SOM - b_1 6$ los tres algoritmos se acercan notablemente a dichos valores para el resto de casos.

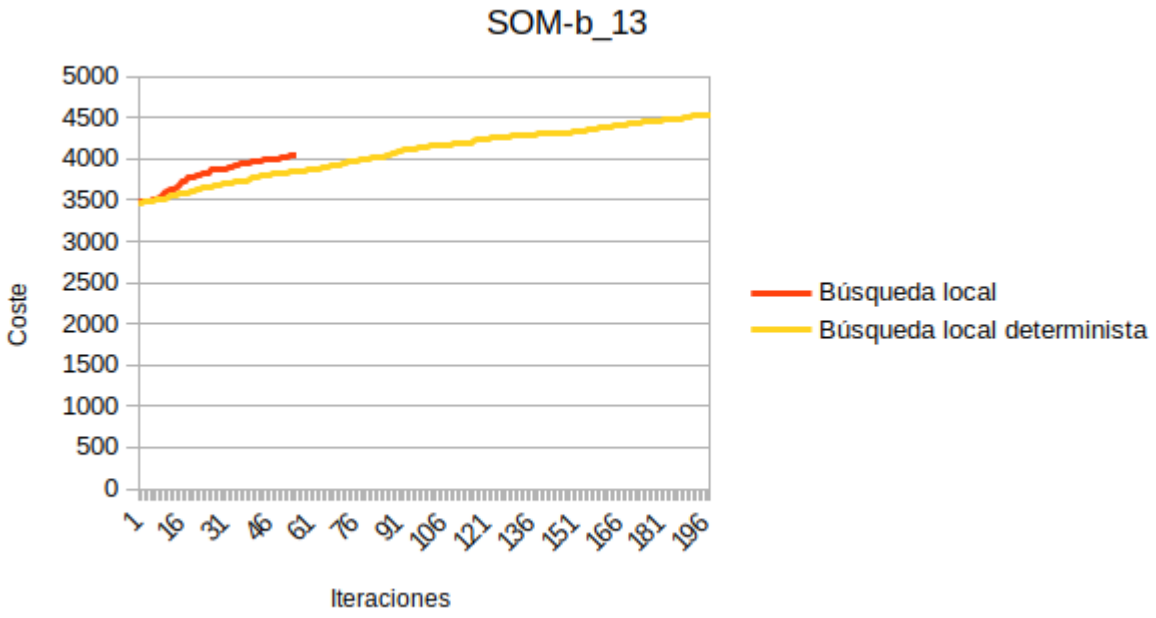
A pesar de los que los costes obtenidos en los distintos algoritmos son sumamente parecidos, no es ni mucho menos el caso para los tiempos y las iteraciones. Se comprueba sin duda la efectividad de la exploración del entorno implementada para la $BLD$ observando los estos valores. Además, se comprueba 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.

## 6.4. 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 $MDG - a_2 1$ y $SOM - b_1 3$, 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.






### 6.4.1. Análisis

Podemos ver como en los distintos casos la evolución es particularmente distinta. Por un lado, para $MDG - a_2 1$ 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 $SOM - b_1 3$ 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.