Carlota Valdivia Manzano - 77158229Z
email: carlotavm99@correo.ugr.es
Doble Grado de Ingeniería Informática y Matemáticas
Curso 2020/21
Grupo 1. Horario Jueves 17:30-19:30

Problema de Máxima Diversidad (MDP):

Técnicas de Búsqueda Local y Algoritmos Greedy

Metaheurísticas: Práctica 1.a, Grupo 1

UNIVERSIDAD DE GRANADA

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 *Sel* de *m* elementos (|*Sel*| = *m*) de un conjunto inicial *S* de *n* elementos, con *n* > *m*, de forma que se **maximice** la diversidad entre los elementos escogidos.

Dicha diversidad se calcula sumando las distancias entre los pares de elementos del subconjunto *Sel*. Para ello se dispone de una matriz D=(d_{ij}) de dimensión nxn que contiene las distancias entre dichos elementos.

La variante del problema a realizar es **MaxSum** (MDP), en la cual la diversidad se calcula como la suma de las distancias entre cada par de elementos seleccionados. Así pues, la definición matemática del problema **MaxSum Diversity Problem** (MDP), es:

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

$$\text{Sujeto a } \sum_{i=1}^{n} x_i = m \text{ con } x_i = \{0, 1\},\ i = 1,\ldots, n.$$

donde x es el vector binario solución al problema.

2. Descripción de la aplicación de los algoritmos empleados

En este apartado describiremos las consideraciones comunes a los distintos algoritmos. Esto incluye la descripción del esquema de representación de soluciones, y la descripción en pseudocódigo de la función objetivo y los operadores comunes a los algoritmos empleados.

En primer lugar, con respecto a la implementación del ćodigo, el **lenguaje de programación** empleado en ambos algoritmos ha sido **C++**., con el uso de algunas librerías con funcionalidades concretas como *<random>, <chrono>* y otras para el uso de estructuras de datos de stl.

Por otro lado, la **entrada de datos** se realiza de igual forma en ambos algoritmos, desde un fichero de texto *.txt,* que se introduce como parámetro, cuyo formato consta de dos números enteros en la primera línea, que hacen referencia a el número de elementos total (n), y al número de elemento seleccionados (m). A continuación, las sucesivas líneas del fichero son dos elementos del total y su distancia entre ellos.

Para almacenar dichos datos en el programa, se ha empleado una matriz (un vector de vectores de float), y como la matriz es simétrica, ha sido suficiente rellenar solamente la triangular superior.

Además se han implementado **otros métodos** para la impresión de la matriz y de otras estructuras de datos, entre ellas set y vector de pares.

2.1 Representación de las soluciones

Para representar las soluciones he optado por utilizar estructuras distintas dependiendo del algoritmo. No obstante para almacenar el valor de la contribución total de la solución he empleado en ambos casos una variable tipo float, que se calcula al final del algoritmo.

```
float solucion;
```

2.1.1 Solución Greedy

Para la solución empleando una estrategia Greedy la representación escogida ha sido una estructura de datos ordenada de enteros, en concreto, un set de enteros.

Cada entero representa un elemento del conjunto total, y dicho elemento va a ser único ya que en dicha estructura no se puede almacenar elementos repetidos. Además los elementos siempre

se van a encontrar ordenados.

El esquema de representación es el siguiente:

```
std::set<int> Sel;
```

2.1.2 Solución Búsqueda Local

Para la solución empleando el algoritmo de Búsqueda Local la representación elegida ha sido un vector de pares de enteros y float. Cada par del vector representa un elemento del total y su contribución, el primer elemento (el entero) representa el número del elemento, y el segundo elemento representa la contribución de dicho elemento en el conjunto de seleccionados Sel.

```
std::vector<pair<int,float> > Sel;
```

2.2 Función objetivo

La implementación de la función objetivo se ha implementado con la misma estructura en ambos algoritmos, a excepción de que la solución está representada en distintas estructuras de datos según el algoritmo empleado.

2.2.1 Función objetivo Greedy

Algorithm 1: solucionMDP

Data: Sel : $set < int >$, matriz: $vector < vector < float >>$
Result: diversidad : float
begin
  diversidad $\leftarrow 0$
  foreach $j \in Sel.v$ **do**
    foreach $j = i \in Sel$ **do**
      diversidad $\leftarrow$ diversidad + matriz[i][j]
    end
  end
end

2.2.2 Función objetivo Búsqueda Local

Algorithm 2: solucionMDP

Data: Sel : $vector < pair < int, float >>$, matriz: $vector < vector < float >>$
Result: diversidad : float
begin
  diversidad $\leftarrow 0$
  foreach $i \in Sel$ **do**
    foreach $j = i \in Sel$ **do**
      diversidad $\leftarrow$ diversidad + matriz[i.first][j.first]
    end
  end
end

3. Algoritmos

Para esta práctica se han implementado dos algoritmos distintos:

- ➔ Algoritmo Greedy-MDP
- ➔ Búsqueda Local

3.1 Algoritmo Greedy-MDP

El **algoritmo Greedy del MDP** se basa en la heurística de ir seleccionando los elementos más lejanos a los previamente seleccionados. Para ello, **partimos del** elemento **más alejado** del resto en el conjunto completo de elementos y en los m-1 siguientes pasos se va escogiendo sin repetir el elemento más lejano a los elementos seleccionados hasta el momento.

Cabe mencionar que este algoritmo no tiene ninguna heurística para solventar los casos en los que hay dos elementos que se encuentran igual de lejos a los demás. En esta situación el algoritmo simplemente se queda con el primero. Además, la solución obtenida va a ser siempre la misma para un mismo conjunto de elementos y distancias.

Para la implementación de este algoritmo se han empleado **tres métodos**. Un método principal denominado “greedy” que engloba todo el algoritmo, y dos métodos auxiliares que se usan reiteradamente en el algoritmo “greedy”.

3.1.1 Método distAc

El método calcula la distancia acumulada de un elemento al resto de elementos del conjunto S.

Algorithm 3: distAc

```
Data: indice : int, S : set < int >, matriz : vector < vector < float >>
Result: distancia : float
begin
    dist ← 0
    foreach i ∈ S do
        if i < indice then
            dist ← dist + matriz[i][indice]
        end
        else if i > indice then
            dist ← dist + matriz[indice][i]
        end
    end
end
```

3.1.2 Método dist

El método calcula la distancia de un elemento no seleccionado a los elementos del conjunto de los seleccionados.

```
Algorithm 4: dist
  Data: indice : int, Sel : set < int >, matriz : vector < vector < float >>
  Result: min : float
  begin
    dist ← 0
    min ← MAX-FLOAT
    foreach i ∈ S do
      if i < indice then
        dist ← matriz[i][indice]
      end
      else if i > indice then
        dist ← matriz[indice][i]
      end
      if dist < min then
        min ← dist
      end
    end
  end
```

3.1.2 Método Greedy

En este método se encuentra implementado en sí el algoritmo Greedy, que llamará las funciones citadas para resolver el problema como se ha descrito anteriormente.

```
Algorithm 5: greedy
  Data: m : int, S : set < int >, Sel : vector < pair < int, float >>, matriz :
        vector < vector < float >>
  begin
    max ← -1
    distMax ← -1
    distActual ← 0
    foreach i ∈ S do
      distActual ← distAc(i, S, matriz)
      if distActual > distMax then
        max ← i
        distMax ← distActual
      end
    end
    Sel ← Sel ∪ max
    S ← S - max
    while Sel.size() < m do
      distMax ← -1
      foreach i ∈ S do
        distActual ← dist(i, Sel, matriz)
        if distActual > distMax then
          max ← i
          distMax ← distActual
        end
        Sel ← Sel ∪ max
        S ← S - max
      end
    end
  end
```

3.2 Búsqueda Local

El **Algoritmo de Búsqueda Local** considerado está basado en el **esquema del primer mejor**. Partimos de una solución generada de forma aleatoria, a la cual calcularemos sus contribuciones y **ordenaremos con un criterio** de ordenación **según la contribución,** los elementos con menor contribución son los que presentan menor diversidad y por tanto son los peores resultados.

Basándonos en el esquema del primer mejor, vamos a ir comparando la diversidad que presenta el elemento más lejano de la solución, frente aquellos elementos no seleccionados. Para compararlos calcularemos el **coste del movimiento** como la diferencia de costes de las dos soluciones factorizadas. Si dicho coste es positivo, significa que el dicho elemento es a priori mejor, y procedemos a realizar un intercambio y a actualizar los valores de las contribuciones calculadas de la solución **de forma factorizada.**

El algoritmo **para** cuando hayamos explorado todos los elementos no seleccionados sin encontrar mejora o cuando se hayan realizado 100000 evaluaciones.

Para implementar esta solución he usado **cinco métodos.** Un método principal llamado "busquedaLocal" donde se realizan todo el proceso explicado anteriormente, llamando a todos los otros submétodos referentes. Dichos métodos son:

3.2.1 Método criterioOrdenacion

Este método es el criterio por el cual se van a ordenar los elementos del vector de pares solución. Aquellos elementos que tengan una menor contribución se situarán primero.

Algorithm 6: criterioOrdenacion

Data: elem1 : $pair < int, float >$, elem2 : $pair < int, float >$

Result: elem1.second $<$ elem2.second

3.2.2 Método solucionAleatoria

En este método se obtiene de forma aleatoria la primera solución del problema, cerciorándonos que no se repitan elementos, y calculamos sus contribuciones posteriormente.

```
Algorithm 7: solucionAleatoria
  Data: m : int, S : set < int >, Sel : vector < pair < int, float >>, matriz :
        vector < vector < float >>
  begin
    n ← matriz[0].size()
    while Sel.size() < m do
      aleatorio.first ← rand()if aleatorio ∈ Sel then
        Sel ← Sel ∪ aleatorio S ← S - aleatorio.first
      end
    end
    foreach i ∈ Sel do
      foreach j ∈ Sel do
        if i.first < j.first then
          i.second ← i.second + matriz[i.first][j.first]
        end
        else if i.first > j.first then
          i.second ← i.second + matriz[j.first][i.first]
        end
      end
    end
    sort(Sel,criterioOrdenacion)
  end
```

3.2.3 Método mejoraIntercambio

Este método nos permite hacer la comparación entre pares a la hora de generar un vecino. Consiste en ver si un elemento de S (no seleccionado) presenta una mejor diversidad a la solución que el peor elemento de la solución.

La variable “diferencia” la pasamos como argumento por referencia para poder guardar el valor en caso de que se produzca intercambio y calcular de forma sencilla la contribución del nuevo elemento como la suma de la del peor elemento con dicha diferencia.

```
Algorithm 8: mejoraIntercambio
  Data: diferencia : float, elemento : int, nuevoElemento : int, Sel :
        vector < pair < int, float >>, matriz: vector < vector < float >>
  Result: mejora : boolean
  begin
    diferencia ← -Sel[elemento].second
    foreach i ∈ Sel do
      if i! = Sel[elemento] then
        if nuevoElemento < i.first then
          diferencia ← diferencia + matriz[ nuevoElemento][i.first]
        end
        else if nuevoElemento > i.first then
          diferencia ← diferencia + matriz[i.first][nuevoElemento]
        end
      end
    end
    if diferencia > 0 then
      mejora ← true
    end
    else
      mejora ← false
    end
  end
```

3.2.4 Método actualizarContribuciones

Este método nos permite actualizar las contribuciones de los todos los elementos de la solución, una vez que hemos encontrado una mejora a algún elemento. Para hacer esa actualización no es necesario calcularlo todo de nuevo, sino a cada elemento le restamos la distancia al elemento que hemos eliminado y le sumamos la distancia al elemento que vamos a introducir.

La contribución del nuevo elemento se calcula en el método “busquedaLocal” de la forma que se ha explicado en el apartado anterior. Una vez actualizado todo, volvemos a ordenar el vector solución.

Algorithm 9: actualizarContribuciones

Data: elementoIn : $pair < int, float >$, elementoOut : $pair < int, float >$, Sel : $vector < pair < int, float >>$, matriz: $vector < vector < float >>$

```
begin
    Sel ← Sel - elementoOut
    foreach i ∈ Sel do
        if i.first < elementoOut.first then
            i.second ← i.second - matriz[i.first][elementoOut.first]
        end
        else if i.first > elementoOut.first then
            i.second ← i.second - matriz[elementoOut.first][i.first]
        end
        if i.first < elementoIn.first then
            i.second ← i.second + matriz[i.first][elementoIn.first]
        end
        else if elementoIn.first > i.first then
            i.second ← i.second + matriz[elementoIn.first][i.first]
        end
    end
    Sel ← Sel ∪ elementoIn
    sort(Sel,criterioOrdenacion)
end
```

3.2.5 Método busquedaLocal

En este método se encuentra implementado en sí el algoritmo de Búsqueda Local, el cual va a ir llamando a todas las funciones citadas anteriormente para poder resolver el problema de la forma que se ha descrito unas páginas atrás.

Algorithm 10: busquedaLocal

```
Data: m : int, S : set < int >, Sel : vector < pair < int, float >>, matriz :
      vector < vector < float >>
begin
    solucionAleatoria(m,S,Sel,matriz)
    cont ← 0
    indice ← 0
    diferencia ← 0
    it ← S.begin()
    while cont < max_it and indice < Sel.size() do
        min ← Sel[indice]
        elementoAct.first ← *it
        elementoAct.second ← 0
        diferencia ← 0
        if mejoraIntercambio(diferencia,indice,*it,Sel,matriz) then
            elementoAct.second ← Sel[indice].second + diferencia
            actualizarContribuciones(elementoAct,min,Sel,matriz)
            S ← S - it
            S ← S + min.first
            indice ← 0
            it ← S.begin()
        end
        else
            it ← it + 1
        end
        if it == S.end() then
            indice ← indice + 1
            it ← S.begin()
        end
        cont ← cont + 1
    end
end
```

3. Procedimiento considerado para desarrollar la práctica

La práctica se ha desarrollado en **C++** y no se ha usado ningún framework de metaheurísticas concreto adicional, puesto que la implementación de los algoritmos no lo ha requerido.

No obstante sí que hemos utilizado distintas **librerías** como *chrono, set, random, vector, limits* y *algorithm.* La librería *chrono* ha sido empleada para medir tiempo, *random* para poder generar de forma aleatoria los elementos de la solución en la Búsqueda Local, *algorithm* para utilizar una función de ordenación de vectores (sort), y librerías como *vector* y *set* para poder utilizar las distintas estructuras de datos mencionadas.

El proyecto se encuentra dividido en **cinco directorios**: *bin, input, output, src* y *zip*. En el directorio raíz se encuentran **cinco archivos:** el *makefile*, tres scripts para ejecutar el algoritmo Greedy, el de Búsqueda Local y ambos a la vez, y un archivo llamado *LEEME.md* con toda la información referente a la estructura del directorio raíz y sus distintos subdirectorios y archivos, y a la ejecución de los programas.

En el directorio src se encuentra el código fuente de ambos algoritmos, que han sido implementados en ficheros *.cpp* distintos. En el directorio *bin* se encuentran los ejecutables de ambos programas. En el directorio input se encuentran los distintos ficheros con los datos que pasaremos como argumento a la hora de ejecutarlos, y en el directorio output se crean archivos con la información de cada ejecución del algoritmo: el fichero usando, el tiempo empleado y la contribución de la solución.

En el **makefile** se han creado tres macros para poder **ejecutar** cada programa, tanto por separado como a la vez. Las macros ***execute_greedy*** y ***execute_bl*** ejecutan por separado todas los ficheros de entrada usando el algoritmo greedy o el búsqueda local, para ejecutar los dos a la vez hay que utilizar la macro ***execute_all***.

Con respecto a la semilla empleada en la Búsqueda Local, se pasa como segundo argumento en la ejecución del programa, por lo que para modificarla en la ejecución haría falta modificar el fichero ***executeBusquedaLocal*** y cambiarle el valor a la variable ***semilla***.

Cabe mencionar que todos los experimentos han sido realizados en una misma ordenador portátil con las siguientes características: Sistema operativo *Ubuntu 18.04.5*, arquitectura *x86_64*, procesador *Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz* y con una memoria RAM de 16GB de tipo DDR4.

4. Experimentos y análisis de resultados

4.1 Descripción de los casos del problema empleados

Se han utilizado 30 casos seleccionados de varios de los conjuntos de instancias disponibles en la MDPLIB, todas pertenecientes al grupo MDG con distancias aleatorias:

- 10 del grupo **MDG-a** con distancias enteras en {0,10}, n=500 y m=50 (*MDG-a_1_n500_m50 a MDG-a_10_n500_m50*)
- 10 del grupo **MDG-b** con distancias reales en [0,1000], n=2000 y m=20 (*MDG-b_21_n2000_m200 a MDG-b_30_n2000_m200*)
- 10 del grupo **MDG-c** con distancias enteras en {0,1000}, n=3000 y m={300, 400, 500, 600} (*MDG-c_1_n3000_m300 a MDG-c_20_n3000_m600*)

4.2 Descripción de los valores de los parámetros considerados en las ejecuciones

En el algoritmo Greedy, el único parámetro considerado a la hora de realizar una ejecución del programa es el fichero con todos los datos referentes a los elementos del conjunto y a las distancias entre dichos elementos. El programa comprueba que el número de parámetros es correcto y procede a realizar la lectura de datos.

En el algoritmo de Búsqueda Local, son necesarios dos parámetros a la hora de ejecutar el programa. El primero es el fichero con todos los datos referentes a los elementos del conjunto y a las distancias entre dichos elementos, y el segundo es el valor de la **semilla** para poder generar números aleatorios.

Para todas las ejecuciones de dicho algoritmo he usado la misma semilla, que ha sido concretamente “2”.

4.3 Resultados obtenidos

A continuación se muestran las tablas con los resultados obtenidos en las ejecuciones de los programas en los distintos casos y algoritmos. En ambas tablas, el tiempo de ejecución se encuentra en milisegundos.

Resultados obtenidos: Algoritmo Greedy

Algoritmo Greedy			
Caso	**Coste obtenido**	**Desv**	**Tiempo (ms)**
MDG-a_1_n500_m50	6865,9400	12,36	4,00
MDG-a_2_n500_m50	6754,0200	13,09	4,00
MDG-a_3_n500_m50	6741,6000	13,12	4,00
MDG-a_4_n500_m50	6841,5900	11,95	4,00
MDG-a_5_n500_m50	6740,3400	13,09	4,00
MDG-a_6_n500_m50	7013,9300	9,77	4,00
MDG-a_7_n500_m50	6637,4600	14,59	4,00
MDG-a_8_n500_m50	6946,2900	10,38	4,00
MDG-a_9_n500_m50	6898,0100	11,22	4,00
MDG-a_10_n500_m50	6853,6800	11,91	4,00
MDG-b_21_n2000_m200	10314500,0000	8,72	253,00
MDG-b_22_n2000_m200	10283300,0000	8,89	254,00
MDG-b_23_n2000_m200	10224200,0000	9,52	250,00
MDG-b_24_n2000_m200	10263600,0000	9,10	252,00
MDG-b_25_n2000_m200	10250100,0000	9,26	251,00
MDG-b_26_n2000_m200	10196200,0000	9,71	259,00
MDG-b_27_n2000_m200	10358100,0000	8,38	256,00
MDG-b_28_n2000_m200	10277400,0000	8,89	252,00
MDG-b_29_n2000_m200	10291300,0000	8,90	255,00
MDG-b_30_n2000_m200	10263800,0000	9,14	257,00
MDG-c_1_n3000_m300	22943100	7,80	990,00
MDG-c_2_n3000_m300	22982300	7,72	953,00
MDG-c_8_n3000_m400	40434600	6,91	1660,00
MDG-c_9_n3000_m400	40488100	6,79	1676,00
MDG-c_10_n3000_m400	40455800	6,95	2061,00
MDG-c_13_n3000_m500	63170700	5,74	2577,00
MDG-c_14_n3000_m500	62817400	6,21	2560,00
MDG-c_15_n3000_m500	63067200	5,86	2579,00
MDG-c_19_n3000_m600	90565500	5,30	3599,00
MDG-c_20_n3000_m600	90602400	5,27	3930,00

Media Desv:	**9,22**
Media Tiempo:	**838,80**

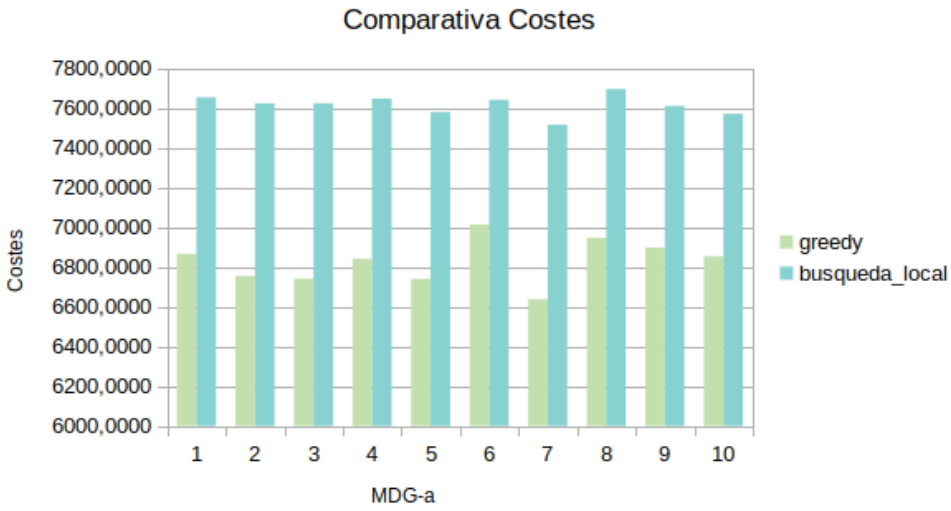
Resultados obtenidos: Algoritmo Búsqueda Local

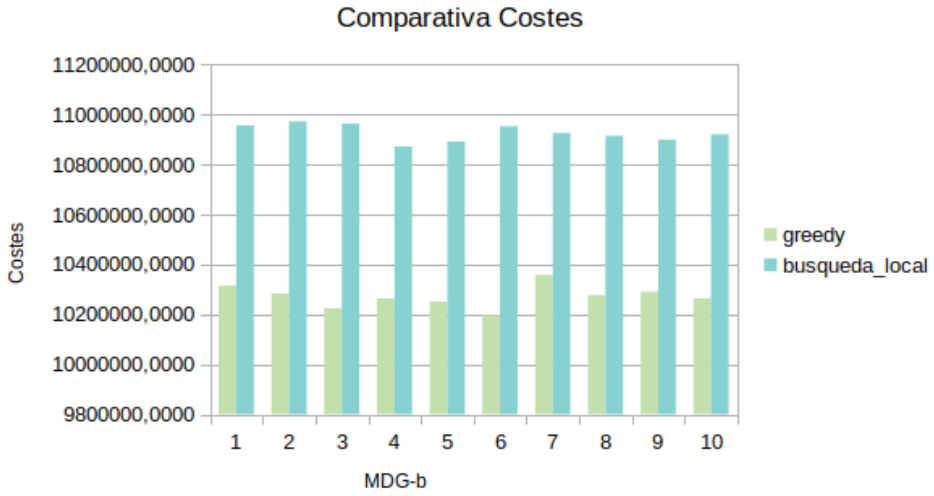
Algoritmo Búsqueda Local			
Caso	**Coste obtenido**	**Desv**	**Tiempo (ms)**
MDG-a_1_n500_m50	7652,8600	2,31	5,00
MDG-a_2_n500_m50	7623,4700	1,91	4,00
MDG-a_3_n500_m50	7623,9400	1,75	3,00
MDG-a_4_n500_m50	7646,4200	1,59	5,00
MDG-a_5_n500_m50	7579,2800	2,27	5,00
MDG-a_6_n500_m50	7641,1500	1,71	5,00
MDG-a_7_n500_m50	7515,5800	3,30	3,00
MDG-a_8_n500_m50	7694,9300	0,72	4,00
MDG-a_9_n500_m50	7609,8600	2,06	4,00
MDG-a_10_n500_m50	7570,5900	2,70	5,00
MDG-b_21_n2000_m200	10957300,0000	3,03	47,00
MDG-b_22_n2000_m200	10973200,0000	2,78	46,00
MDG-b_23_n2000_m200	10964400,0000	2,97	45,00
MDG-b_24_n2000_m200	10872400,0000	3,71	46,00
MDG-b_25_n2000_m200	10892300,0000	3,57	48,00
MDG-b_26_n2000_m200	10953400,0000	3,00	45,00
MDG-b_27_n2000_m200	10926600,0000	3,35	47,00
MDG-b_28_n2000_m200	10915200,0000	3,23	46,00
MDG-b_29_n2000_m200	10899600,0000	3,52	48,00
MDG-b_30_n2000_m200	10921700,0000	3,32	49,00
MDG-c_1_n3000_m300	23999000	3,56	79,00
MDG-c_2_n3000_m300	23983300	3,70	77,00
MDG-c_8_n3000_m400	42015800	3,27	105,00
MDG-c_9_n3000_m400	41903100	3,53	106,00
MDG-c_10_n3000_m400	41965300	3,48	106,00
MDG-c_13_n3000_m500	64966700	3,06	133,00
MDG-c_14_n3000_m500	64945200	3,04	137,00
MDG-c_15_n3000_m500	64981600	3,00	139,00
MDG-c_19_n3000_m600	93050700	2,70	164,00
MDG-c_20_n3000_m600	93112300	2,65	165,00

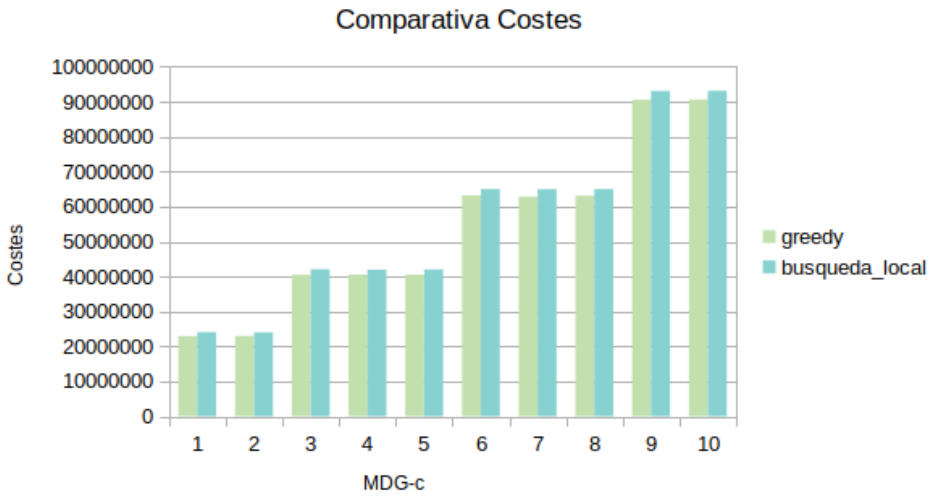
Media Desv:	**2,83**
Media Tiempo:	**57,37**

4.4 Comparativas

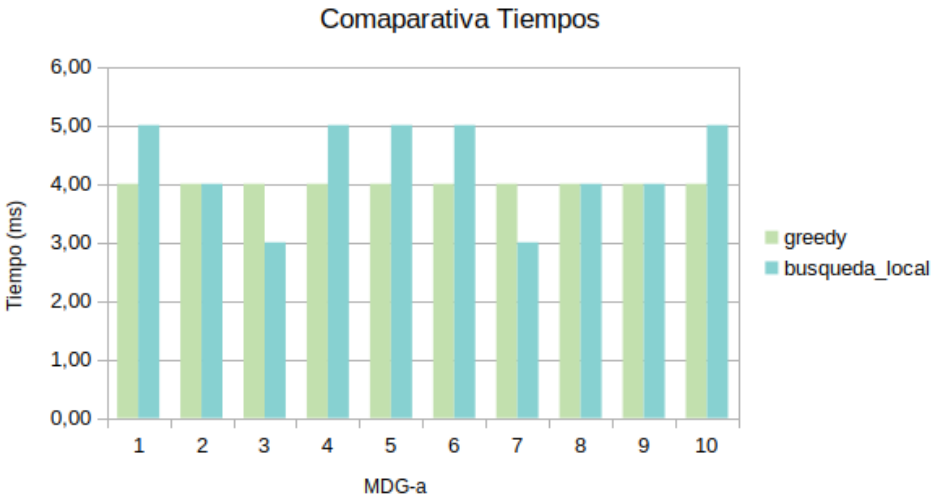
Comparativa Diversidad		
Caso	**Greedy**	**Búsqueda Local**
MDG-a_1_n500_m50	6865,9400	7652,8600
MDG-a_2_n500_m50	6754,0200	7623,4700
MDG-a_3_n500_m50	6741,6000	7623,9400
MDG-a_4_n500_m50	6841,5900	7646,4200
MDG-a_5_n500_m50	6740,3400	7579,2800
MDG-a_6_n500_m50	7013,9300	7641,1500
MDG-a_7_n500_m50	6637,4600	7515,5800
MDG-a_8_n500_m50	6946,2900	7694,9300
MDG-a_9_n500_m50	6898,0100	7609,8600
MDG-a_10_n500_m50	6853,6800	7570,5900
MDG-b_21_n2000_m200	10314500,0000	10957300,0000
MDG-b_22_n2000_m200	10283300,0000	10973200,0000
MDG-b_23_n2000_m200	10224200,0000	10964400,0000
MDG-b_24_n2000_m200	10263600,0000	10872400,0000
MDG-b_25_n2000_m200	10250100,0000	10892300,0000
MDG-b_26_n2000_m200	10196200,0000	10953400,0000
MDG-b_27_n2000_m200	10358100,0000	10926600,0000
MDG-b_28_n2000_m200	10277400,0000	10915200,0000
MDG-b_29_n2000_m200	10291300,0000	10899600,0000
MDG-b_30_n2000_m200	10263800,0000	10921700,0000
MDG-c_1_n3000_m300	22943100	23999000
MDG-c_2_n3000_m300	22982300	23983300
MDG-c_8_n3000_m400	40434600	42015800
MDG-c_9_n3000_m400	40488100	41903100
MDG-c_10_n3000_m400	40455800	41965300
MDG-c_13_n3000_m500	63170700	64966700
MDG-c_14_n3000_m500	62817400	64945200
MDG-c_15_n3000_m500	63067200	64981600
MDG-c_19_n3000_m600	90565500	93050700
MDG-c_20_n3000_m600	90602400	93112300

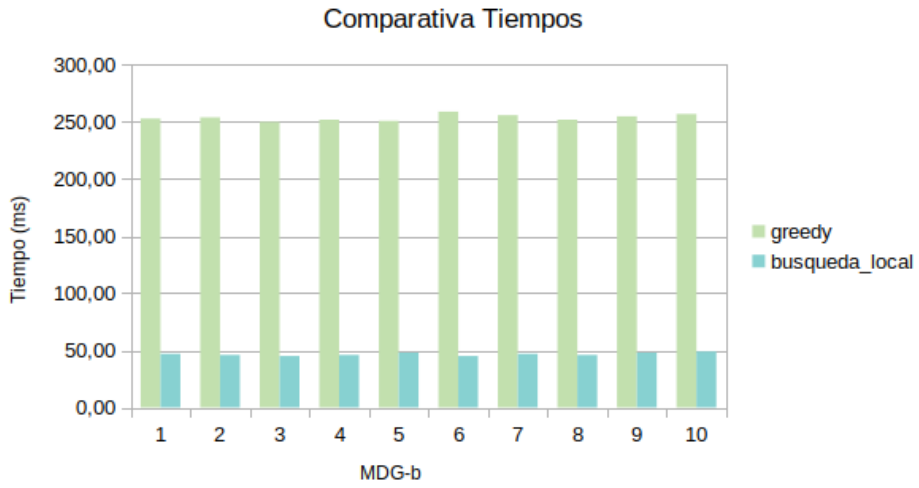

Comparativa Costes
7800,0000
7600,0000
7400,0000
7200,0000
7000,0000
6800,0000
6600,0000
6400,0000
6200,0000
6000,0000
Costes
1
2
3
4
5
6
7
8
9
10
MDG-a
greedy
busqueda_local

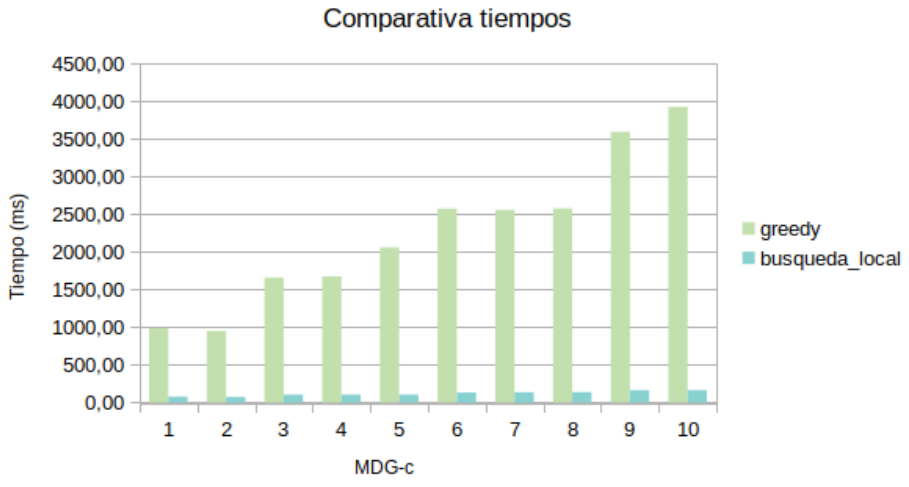

Comparativa Costes
11200000,0000
11000000,0000
10800000,0000
10600000,0000
10400000,0000
10200000,0000
10000000,0000
9800000,0000
Costes
1
2
3
4
5
6
7
8
9
10
MDG-b
greedy
busqueda_local


Comparativa Costes
100000000
90000000
80000000
70000000
60000000
50000000
40000000
30000000
20000000
10000000
0
Costes
1
2
3
4
5
6
7
8
9
10
MDG-c
greedy
busqueda_local

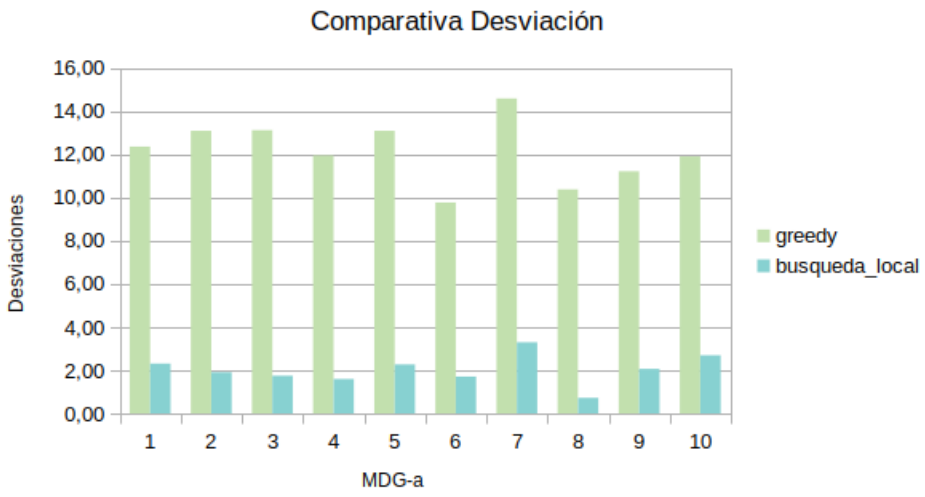
Comparativa Tiempo (ms)		
Caso	**Greedy**	**Búsqueda Local**
MDG-a_1_n500_m50	4,00	5,00
MDG-a_2_n500_m50	4,00	4,00
MDG-a_3_n500_m50	4,00	3,00
MDG-a_4_n500_m50	4,00	5,00
MDG-a_5_n500_m50	4,00	5,00
MDG-a_6_n500_m50	4,00	5,00
MDG-a_7_n500_m50	4,00	3,00
MDG-a_8_n500_m50	4,00	4,00
MDG-a_9_n500_m50	4,00	4,00
MDG-a_10_n500_m50	4,00	5,00
MDG-b_21_n2000_m200	253,00	47,00
MDG-b_22_n2000_m200	254,00	46,00
MDG-b_23_n2000_m200	250,00	45,00
MDG-b_24_n2000_m200	252,00	46,00
MDG-b_25_n2000_m200	251,00	48,00
MDG-b_26_n2000_m200	259,00	45,00
MDG-b_27_n2000_m200	256,00	47,00
MDG-b_28_n2000_m200	252,00	46,00
MDG-b_29_n2000_m200	255,00	48,00
MDG-b_30_n2000_m200	257,00	49,00
MDG-c_1_n3000_m300	990,00	79,00
MDG-c_2_n3000_m300	953,00	77,00
MDG-c_8_n3000_m400	1660,00	105,00
MDG-c_9_n3000_m400	1676,00	106,00
MDG-c_10_n3000_m400	2061,00	106,00
MDG-c_13_n3000_m500	2577,00	133,00
MDG-c_14_n3000_m500	2560,00	137,00
MDG-c_15_n3000_m500	2579,00	139,00
MDG-c_19_n3000_m600	3599,00	164,00
MDG-c_20_n3000_m600	3930,00	165,00
Media Tiempo:	**838,80**	**57,37**

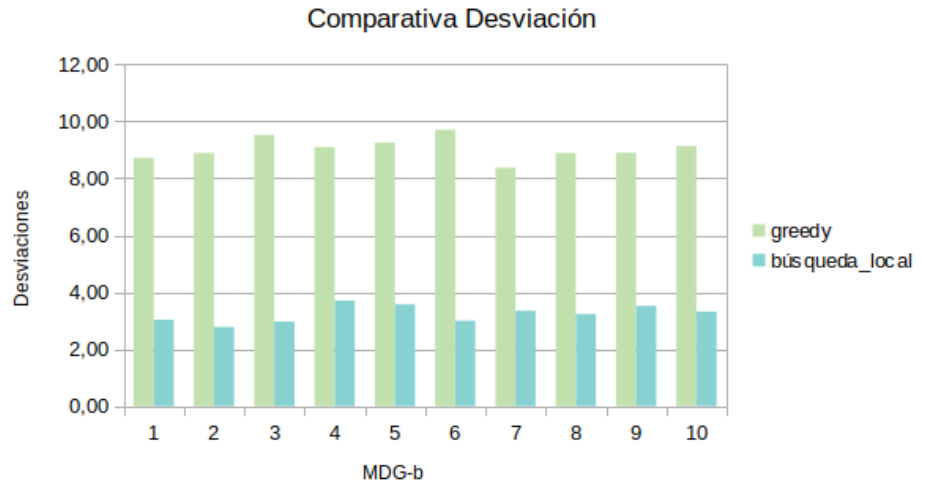

Comaparativa Tiempos
6,00
5,00
4,00
3,00
2,00
1,00
0,00
Tiempo (ms)
1
2
3
4
5
6
7
8
9
10
MDG-a
greedy
busqueda_local

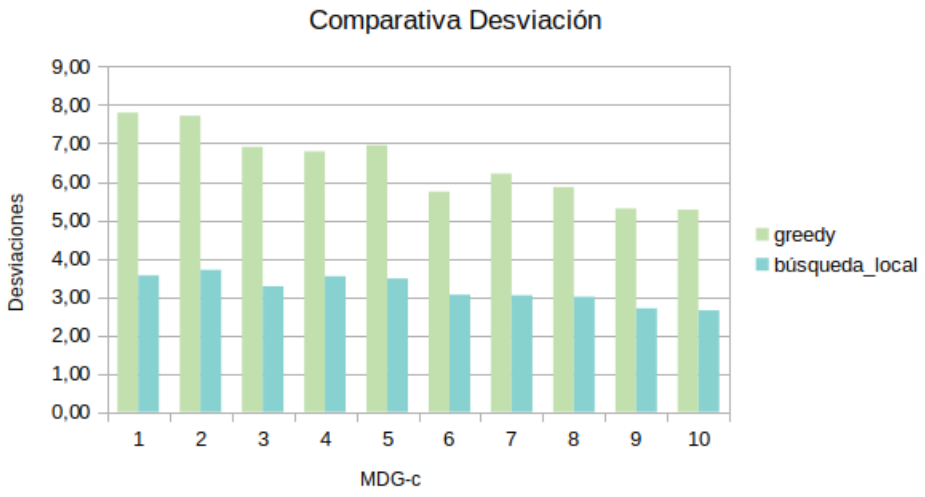

Comparativa Tiempos
300,00
250,00
200,00
150,00
100,00
50,00
0,00
Tiempo (ms)
1
2
3
4
5
6
7
8
9
10
MDG-b
greedy
busqueda_local


Comparativa tiempos
4500,00
4000,00
3500,00
3000,00
2500,00
2000,00
1500,00
1000,00
500,00
0,00
Tiempo (ms)
1
2
3
4
5
6
7
8
9
10
MDG-c
greedy
busqueda_local

Comparativa Desviación		
Caso	**Greedy**	**Búsqueda Local**
MDG-a_1_n500_m50	12,36	2,31
MDG-a_2_n500_m50	13,09	1,91
MDG-a_3_n500_m50	13,12	1,75
MDG-a_4_n500_m50	11,95	1,59
MDG-a_5_n500_m50	13,09	2,27
MDG-a_6_n500_m50	9,77	1,71
MDG-a_7_n500_m50	14,59	3,30
MDG-a_8_n500_m50	10,38	0,72
MDG-a_9_n500_m50	11,22	2,06
MDG-a_10_n500_m50	11,91	2,70
MDG-b_21_n2000_m200	8,72	3,03
MDG-b_22_n2000_m200	8,89	2,78
MDG-b_23_n2000_m200	9,52	2,97
MDG-b_24_n2000_m200	9,10	3,71
MDG-b_25_n2000_m200	9,26	3,57
MDG-b_26_n2000_m200	9,71	3,00
MDG-b_27_n2000_m200	8,38	3,35
MDG-b_28_n2000_m200	8,89	3,23
MDG-b_29_n2000_m200	8,90	3,52
MDG-b_30_n2000_m200	9,14	3,32
MDG-c_1_n3000_m300	7,80	3,56
MDG-c_2_n3000_m300	7,72	3,70
MDG-c_8_n3000_m400	6,91	3,27
MDG-c_9_n3000_m400	6,79	3,53
MDG-c_10_n3000_m400	6,95	3,48
MDG-c_13_n3000_m500	5,74	3,06
MDG-c_14_n3000_m500	6,21	3,04
MDG-c_15_n3000_m500	5,86	3,00
MDG-c_19_n3000_m600	5,30	2,70
MDG-c_20_n3000_m600	5,27	2,65
Media Desv:	**9,22**	**2,83**


Comparativa Desviación
16,00
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
Desviaciones
1
2
3
4
5
6
7
8
9
10
MDG-a
greedy
busqueda_local


Comparativa Desviación
12,00
10,00
8,00
6,00
4,00
2,00
0,00
Desviaciones
1
2
3
4
5
6
7
8
9
10
MDG-b
greedy
búsqueda_local


Comparativa Desviación
9,00
8,00
7,00
6,00
5,00
4,00
3,00
2,00
1,00
0,00
Desviaciones
1
2
3
4
5
6
7
8
9
10
MDG-c
greedy
búsqueda_local

4.5 Análisis de resultados

Para realizar un análisis de resultados partimos de dos tablas de datos con los costes, desviaciones y tiempo de ejecución de cada algoritmo para unos datos distintos. Dichas tablas nos dan por sí solas información al respecto de cada algoritmo programado, podemos ver como varían los costes según las desviaciones y ver el tiempo de ejecución empleado.

A partir de dichas tablas, se ha realizado una comparativa entre ambos algoritmos para poder ver ventajas e inconvenientes de cada uno de ellos. Dichas comparativas se han hecho a raíz de tres tablas distintas donde se comparan cada una de las columnas de las dos tablas iniciales. Para una mejor comprensión y visualización se han creado para cada nueva tabla comparativa tres gráficas distintas según el tipo de datos de entrada.

A continuación, procederemos a valorar de forma individual cada algoritmo a partir de los resultados obtenidos, después se hará una valoración comparando ambos.

Para empezar, podemos observar que en la primera tabla, la del algoritmo **Greedy,** nos proporciona una solución aproximada y moderadamente correcta. Los algoritmos Greedy se caracterizan por son más sencillos de diseñar y por optar por una solución relativamente cercana sin llegar a ser óptima. En este caso, podemos apreciar que efectivamente las desviaciones no son lo suficientemente pequeñas como para poder asegurar que los resultados obtenidos con este algoritmo rozan la optimidad.

No obstante, se puede apreciar que los resultados obtenidos van mejorando según aumenta el tamaño del problema, ya que por ejemplo los mejores resultados son aquellos de los caso MDG-C. También se puede apreciar que se ve una mejora cuando no solo aumenta en el proporción la n y la m, sino cuando aumenta solamente la m, es decir, tenemos que seleccionar un mayor número de elementos del total. La causa de esta mejora podría estar relacionada con que cuanto más elementos se vayan a seleccionar del total es más probable haber escogido los 10 mejores de 20, que los 2 mejores de 20.

Con respecto a los **tiempos del Greedy** se puede observar que en los caso MDG-A los tiempos son relativamente bajos, pero a medida que aumenta el tamaño del problema los tiempos van aumentando hasta llegar a tardar casi 4 segundos en resolver los mapas de mayor tamaño.

A continuación, observamos que en la segunda tabla, la del algoritmo de **Búsqueda Local,** las soluciones obtenidas son bastantes más próximas a la óptima, obteniendo desviaciones que no llegan al 4. Sin embargo, no observamos ningún patrón específico en cuanto a la mejora de soluciones al contrario que el algoritmo Greedy, todas ellas son relativamente buenas con desviaciones entre mayores que 0 y menores que 4. Por otro lado, los tiempos son bastante

mejores con una media de 57.37ms, ya que hemos optado por una versión de la Búsqueda Local empleando el método de factorización lo cual nos permite ahorrar mucho tiempo y cálculos.

Ahora realizaremos una pequeña **comparación** entre los resultados obtenidos entre ambos algoritmos. Podemos apreciar que el algoritmo de Búsqueda Local nos proporciona unos mejores resultados tanto a nivel de eficiente como a nivel de optimalidad. Los resultados tienen mejores desviaciones y mejores tiempos de media, y en la mayoría de casos se cumple.

Para algunos casos, como son los de MDG-A, los tiempos obtenidos en ambos algoritmos son similares, y se podría decir que para tamaños más pequeños las diferencias de tiempo entre el Greedy y la Búsqueda Local son parecidos. Esto puede deberse a que en en estos casos lo normal es que al tener un menor tamaño la Búsqueda Local termine porque ya se ha estudiado todo el vecindario y no por haber llegado al tope de iteraciones permitidas.

En los casos MDG-B y MDG-C las diferencias son bastante considerables entre ambos algoritmos, ya que además de ahorrarnos muchos cálculos en la Búsqueda Local, pararemos antes al llegar al tope de iteraciones permitidas.

Con respecto a la desviaciones todas ellas son notablemente mejores en el algoritmo de Búsqueda Local frente al Greedy, con una diferencia media de más de 6 unidades. La única apreciación que se puede considerar es que para los casos MDG-C la diferencia entre desviaciones es menor, ya que el Greedy proporcionaba mejores resultados.

Como conclusión final, podemos decir que en cuanto a eficiencia y optimalidad las soluciones y tiempos obtenidos con el algoritmo de Búsqueda Local son notablemente mejores que las obtenidas con el algoritmo Greedy. No obstante, a favor del algoritmo Greedy cabe destacar que la dificultad a la hora de programar el algoritmo Greedy es menor que la del segundo algoritmo, pero de forma global podríamos concluir que para este problema los resultados obtenidos con el algoritmo de Búsqueda Local han sido mejores.