UNIVERSIDAD DE GRANADA

Metaheurísticas

Práctica 1.a

Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema de la Máxima Diversidad

Nombre: Francisco Javier Bolívar Expósito

DNI: ---

Email: ---

Grupo de prácticas: ---

Curso 2019-2020

Índice

1. Descripción del Problema

El problema de la máxima diversidad (**MDP**) consiste en **seleccionar** un **subconjunto** de *m* elementos de un conjunto más grande de *n* de forma que la **diversidad** entre los elementos escogidos se **maximice**.

Para calcular la diversidad consideraremos la variante del problema ***MaxSum***, en la que la diversidad se calcula como la suma de las distancias entre cada par de elementos seleccionados.

Por lo tanto el problema consistirá en elegir *m* elementos del conjunto inicial de *n* elementos de forma que la suma de las distancias entre ellos sea máxima, siendo la definición matemática:

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

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

$$x_i = \{0, 1\}, \quad i = 1, \ldots, n.$$

Xi contiene un 1 si el elemento "i" se ha seleccionado y un 0 si no

MDP es de la clase de problemas **NP-Duros** de optimización combinatoria, por lo que **no es factible** encontrar la solución óptima en un tiempo razonable. Por ello en esta práctica se desarrollan y analizan algoritmos eficientes para encontrar soluciones buenas aunque no sean las óptimas.

2. Aplicación de los algoritmos empleados al problema

La práctica consiste en aplicar dos algoritmos al MDP, además se ha programado una variante adicional.

El primero es un algoritmo ***Greedy*** basado en "Añadir secuencialmente el elemento no seleccionado que más diversidad aporte con respecto a los ya seleccionados", seguiremos esta heurística para aplicar el algoritmo.

El segundo es la **búsqueda local del primer mejor** que explora las soluciones vecinas a la actual hasta encontrar una mejor y entonces se cambia a esta. Para adaptarlo mejor al problema y reducir significativamente el tiempo de ejecución utilizaremos una exploración inteligente del vecindario que

genera primero los vecinos más prometedores. Esta técnica la explicaremos más en profundidad posteriormente. Pero antes vamos a ver varios aspectos comunes a los algoritmos.

La variante adicional es una combinación de los dos anteriores que describiremos posteriormente.

2.1. Esquema de representación de soluciones

Se ha usado un esquema de representación de soluciones con números enteros por ser más cómoda y eficiente que la representación binaria presentada en la definición matemática para la implementación de los algoritmos.

Siendo n = número de elementos en el conjunto inicial, m = número de elementos a seleccionar, la solución se representa como un vector de números enteros que:

- Contiene exactamente m elementos
- No contiene elementos repetidos,
- Cada elemento es un entero perteneciente al intervalo [0, n).

Por lo tanto tenemos un vector solución que contiene los índices de los elementos seleccionados.

Para un problema con n = 10 elementos en el conjunto inicial y m = 5 elementos a seleccionar un vector que represente una posible solución sería por ejemplo: Sol = {2, 4, 7, 8, 9}

2.2. Función objetivo

La función objetivo devuelve un valor para un vector solución. El objetivo de los algoritmos de la práctica es maximizar esta función objetivo, que consiste en calcular la suma de distancias entre pares de elementos seleccionados. A continuación se muestra en pseudocódigo la función objetivo:

FuncionObjetivo(Entradas: distancias entre elementos y vector solución “Sol”
Salida: Coste de la solución):
1. Coste = 0
Para i desde 0 hasta m - 1:
Para j desde i+1 hasta m:
2. Coste := Coste + distancia entre Sol_i y Sol_j
devolver Coste

3. Descripción de la implementación de los Algoritmos

3.1. Greedy

Para aplicar el algoritmo descrito anteriormente que quiere " Añadir secuencialmente el elemento no seleccionado que más diversidad aporte con respecto a los ya seleccionados" seguiremos el siguiente procedimiento:

1. Seleccionamos el elemento que más distancia acumulada tenga al resto de elementos
2. Hasta seleccionar m elementos, eligiendo el elemento de los restantes sin seleccionar que más distancia acumulada tenga al conjunto de seleccionados.

A continuación se muestra el pseudocódigo del algoritmo greedy implementado:

Greedy(Entradas: matriz de distancias entre elementos y cantidad de elementos a seleccionar "m"
Salida: Conjunto de elementos seleccionados "*Sel*")

1. *Sel* = Conjunto Vacío
2. Array *distAc* = suma de las filas de la matriz de distancias. Por lo tanto en $distAc_i$ se encontrará la distancia acumulada del elemento "i" al resto de elementos.
3. Buscamos la mayor distancia acumulada de *distAc* y guardamos su índice en *ult_sel*
4. Añadimos a *Sel* el índice guardado en *ult_sel*
5. *distancias[0..m] = 0*

while (|*Sel*| < m):

6. Actualizamos *distancias* que contiene las distancias de los elementos al conjunto de elementos seleccionados *sumándo* las distancias entre el último elemento seleccionado indicado en *ult_sel* y el resto de elementos. Por lo tanto en $distancias_i$ se encontrará la distancia del elemento "i" al resto de elementos seleccionados.
7. Por cada i en *Sel* $distancias_i = 0$
8. *ult_sel* = índice del valor más grande guardado en *distancias*
9. $distancias_{ult\_sel} = 0$
10. Añadimos a *Sel* el elemento *ult_sel*

end while

devolver *Sel*

Greedy aplicado a MDP

3.2. Búsqueda Local

Para generar los vecinos del entorno de la solución actual utilizaremos un **operador de intercambio** consistente en intercambiar un elemento seleccionado por un elemento no seleccionado, si la solución obtenida es mejor pasaremos a esta y si no seguiremos intercambiando.

Utilizaremos una **exploración inteligente del vecindario** para reducir el tiempo de ejecución focalizando la BL en una zona del espacio de búsqueda en la que es más probable que se produzca un cambio de solución. Para esto definiremos un **orden de aplicación** del operador de intercambio.

El orden a seguir consiste en intercambiar el elemento seleccionado que menos contribuye al coste de la función objetivo sucesivamente con los elementos no seleccionados. Si se encuentra un movimiento de mejora se aplica. Si no se encuentra mejora, se pasa al siguiente elemento con menor contribución y se repite el proceso. Si no se encuentra mejora en todo el vecindario se finaliza la ejecución de la BL y se devuelve la solución actual.

La **contribución** de un elemento seleccionado a la solución se define como la suma de distancias entre este elemento y el resto de elementos seleccionados.

$$d_i = \sum_{s_j \in Sel} d_{ij} = d(s_i, Sel)$$

En vez de calcular la función objetivo para cada solución vecina y comprobar si esta mejora la solución actual utilizaremos una **Factorización del Movimiento de Intercambio** para evitar recalcular todas las distancias de la función objetivo.

Si la contribución del elemento añadido a la solución es mayor que la contribución del elemento eliminado en el intercambio entonces la solución vecina será mejor y podemos cambiar a ella.

Si cambiamos a la solución vecina actualizaremos las contribuciones del resto de elementos seleccionados eliminado la distancia de cada uno de ellos al elemento seleccionado y añadiendo la distancia de cada uno de ellos al elemento añadido.

El pseudocódigo para generar aleatoriamente la solución inicial (se han usado funciones de numpy que permiten hacerlo en solo 2 pasos) es el siguiente:

1. Establecer la semilla para el generador aleatorio como “510”
2. Solución inicial = Generar vector de tamaño m, con números en [0, m) sin números repetidos

Con estas consideraciones y la solución inicial obtenida aleatoriamente en la siguiente página se muestra el pseudocódigo del algoritmo de Búsqueda Local Primero el Mejor implementado.

Entrada: “dist” Matriz nxn de distancias entre cada par de elementos
Entrada: “m” Numero de elementos a seleccionar
Entrada: “Sel” Solución inicial, conjunto de elementos seleccionados
Devuelve: “Sel” Conjunto de elementos seleccionados

BusquedaLocal:

1. Calculamos “s” el conjunto de elementos no seleccionados como la diferencia entre el conjunto de todos los elementos y “sel”
2. Realizamos el cálculo de la contribución de cada elemento seleccionado. Siendo Z un vector de tamaño m inicializado a 0. Z_i = suma de las distancias entre el elemento Sel_i y el resto de elementos de *Sel*.

while(evaluaciones de soluciones vecinas < 100000):

3. Barajamos el orden de los elementos no seleccionados

Para cada “*índice*” en una lista de índices según las contribuciones de Z de menor a mayor:

Para cada “elemento” en los elementos no seleccionados:

4. Incrementamos las evaluaciones de soluciones vecinas
5. Intercambiamos $Sel_{índice}$ por *elemento*
6. ContribucionVecino = suma de distancias entre *elemento* y el resto de elementos en *Sel*.

Si ContribucionVecino > Contribucion del elemento eliminado

7. $Z_i = Z_i$ + dist[*elemento*, Sel_i] - dist[*elemento eliminado*, Sel_i]
8. Al haber encontrado mejora vecina volvemos al paso 3.

En caso contrario

9. Volvemos a la solución original deshaciendo el intercambio de Sel_{indice} por elemento

10. Si hemos explorado todas las soluciones vecinas sin encontrar mejora, terminamos el bucle while pasando al paso 11.

11. devolver *Sel*

Búsqueda Local Primero el Mejor aplicada a MDP

3.3. Greedy + BLPM

El Greedy nos devuelve una solución de calidad, no es óptima pero es buena y se consigue en un tiempo despreciable. La BLPM devuelve una solución mejor pero tiene un tiempo de ejecución mucho mayor al Greedy.

Teniendo en cuenta estos resultados se ha probado una combinación de las dos técnicas. En vez de establecer como solución inicial en la Búsqueda Local un punto aleatorio del espacio de búsqueda usamos como solución incial el resultado del algoritmo Greedy.

La esperanza de este intento es que al devolver el algoritmo Greedy soluciones de calidad podemos llegar hasta el óptimo local de forma relativamente rápida aplicando BLPM, consiguiendo así mejorar la solución obtenida por Greedy pero con un tiempo menor que con la aplicación original de BLPM. En los análisis de los resultados veremos que conclusiones sacamos de este algoritmo.

4. Implementación de la práctica y manual de usuario

La implementación de la práctica se ha realizado en **Python**, usando como interprete para las pruebas Python 3.8.1.

Se han utilizado 3 módulos, **time** para medir los tiempos de ejecución, **os** para recorrer el directorio de datos y **numpy** para realizar la mayoría de operaciones, siendo este último el único no incluido en Python.

Se ha evitado en lo máximo posible el uso de bucles y se ha intentado realizar las operaciones que más tiempo consumen del programa con funciones de NumPy, ya sean sumas de filas de matrices, costrucción de arrays con expresiones usando np.fromiter() o sumas, restas de arrays, intentando obtener así una buena eficiencia.

En la implementación se han creado 4 funciones en un solo script.

readDistances(path)

Para leer la matriz de distancias de un fichero de la biblioteca MDPLIB

Entrada: path Ruta del fichero

Entrada: dist Matriz NxM de distancias entre cada par de elementos

Devuelve: m Numero de elementos a seleccionar

fObjetivo(dist, sel)

Calcula el coste de una solución para MDP siguiendo el modelo MaxSum

Entrada: dist Matriz NxM de distancias entre cada par de elementos

Entrada: sel Lista de elementos seleccionados

Devuelve: f Coste obtenido en la función que se quiere maximizar (suma de distancias entre pares de elementos seleccionados)

greedyMDP(dist, m)

Calcula una solución con el algoritmo greedy para el MDP variante MaxSum

Entrada: dist Matriz NxM de distancias entre cada par de elementos

Entrada: m Numero de elementos a seleccionar

Devuelve: sel Conjunto de elementos seleccionados

blpmMDP(sel, dist, m)

Calcula una solución con Busqueda Local Primero el Mejor para el MDP variante MaxSum

Entrada: dist Matriz NxM de distancias entre cada par de elementos

Entrada: m Numero de elementos a seleccionar

Entrada: sel Solución inicial, conjunto de elementos seleccionados

Devuelve: sel Conjunto de elementos seleccionado

Finalmente el script de python hace uso de estas funciones iterando por cada fichero de MDPLIB con los datos de un caso del problema, haciendo para cada uno:

- Lectura de los datos almacenando la cantidad de elementos a seleccionar y una matriz de distancias nxn
- Ejecución del algoritmo Greedy y muestra del coste obtenido y del tiempo de ejecución
- Ejecución del algoritmo de Búsqueda Local y muestra del coste obtenido y del tiempo de ejecución

Para usar el programa es necesario instalar el módulo NumPy. Se incluye un archivo **requirements.txt** que se puede usar para instalar la versión usada, utilizando “pip” con ‘pip install -r requirements.txt’. También se puede instalar usando “pip” con ‘pip install numpy’.

Para la prueba solo será necesario ejecutar el script **MDP.py** que cuelga del directorio **FUENTES** con un interprete de Python, y este leerá y ejecutara todos los casos que se encuentren en el directorio BIN.

5. Análisis de los resultados

Para todos los algoritmos que usan búsqueda local se han usado:

- 100.000 evaluaciones de la función objetivo como máximo
- Semilla del generador aleatorio de NumPy 510.

A continuación se presentan los resultados en el formato especificado, los tiempos se muestran en mili-segundos para facilitar la comparación:

Caso	*n*	*m*	Mejor coste
GKD-c_11_n500_m50	500	50	19587,12891
GKD-c_12_n500_m50	500	50	19360,23633
GKD-c_13_n500_m50	500	50	19366,69922
GKD-c_14_n500_m50	500	50	19458,56641
GKD-c_15_n500_m50	500	50	19422,15039
GKD-c_16_n500_m50	500	50	19680,20898
GKD-c_17_n500_m50	500	50	19331,38867
GKD-c_18_n500_m50	500	50	19461,39453
GKD-c_19_n500_m50	500	50	19477,32813
GKD-c_20_n500_m50	500	50	19604,84375
MDG-b_1_n500_m50	500	50	778030,625
MDG-b_2_n500_m50	500	50	779963,6875
MDG-b_3_n500_m50	500	50	776768,4375
MDG-b_4_n500_m50	500	50	775394,625
MDG-b_5_n500_m50	500	50	775611,0625
MDG-b_6_n500_m50	500	50	775153,6875
MDG-b_7_n500_m50	500	50	777232,875
MDG-b_8_n500_m50	500	50	779168,75
MDG-b_9_n500_m50	500	50	774802,1875
MDG-b_10_n500_m50	500	50	774961,3125
MDG-a_31_n2000_m200	2000	200	114139
MDG-a_32_n2000_m200	2000	200	114092
MDG-a_33_n2000_m200	2000	200	114124
MDG-a_34_n2000_m200	2000	200	114203
MDG-a_35_n2000_m200	2000	200	114180
MDG-a_36_n2000_m200	2000	200	114252
MDG-a_37_n2000_m200	2000	200	114213
MDG-a_38_n2000_m200	2000	200	114378
MDG-a_39_n2000_m200	2000	200	114201
MDG-a_40_n2000_m200	2000	200	114191

Algoritmo Greedy			
Caso	Coste obtenido	Desv	Tiempo
GKD-c_11_n500_m50	19583,8628	0,02	0,85902214
GKD-c_12_n500_m50	19349,1687	0,06	0,83327293
GKD-c_13_n500_m50	19348,8037	0,09	0,91838837
GKD-c_14_n500_m50	19458,3703	0,00	0,85377693
GKD-c_15_n500_m50	19420,2620	0,01	0,84328651
GKD-c_16_n500_m50	19677,3283	0,01	0,82945824
GKD-c_17_n500_m50	19331,3884	0,00	0,90837479
GKD-c_18_n500_m50	19461,3946	0,00	0,83518028
GKD-c_19_n500_m50	19472,8163	0,02	0,88715553
GKD-c_20_n500_m50	19604,8436	0,00	0,85544586
MDG-b_1_n500_m50	753517,3900	3,15	0,99444389
MDG-b_2_n500_m50	754034,9600	3,32	0,92697144
MDG-b_3_n500_m50	759842,6000	2,18	0,81682205
MDG-b_4_n500_m50	757355,4200	2,33	0,87690353
MDG-b_5_n500_m50	763215,8800	1,60	0,86283684
MDG-b_6_n500_m50	753856,6500	2,75	0,8327961
MDG-b_7_n500_m50	756626,3100	2,65	0,89502335
MDG-b_8_n500_m50	757362,1300	2,80	0,81896782
MDG-b_9_n500_m50	759993,6300	1,91	0,83780289
MDG-b_10_n500_m50	754165,7200	2,68	0,90503693
MDG-a_31_n2000_m2	112505	1,43	9,85693932
MDG-a_32_n2000_m2	112425	1,46	9,79661942
MDG-a_33_n2000_m2	112375	1,53	10,1058483
MDG-a_34_n2000_m2	112262	1,70	8,86535645
MDG-a_35_n2000_m2	112558	1,42	8,72731209
MDG-a_36_n2000_m2	112355	1,66	9,36532021
MDG-a_37_n2000_m2	112453	1,54	9,0637207
MDG-a_38_n2000_m2	112495	1,65	9,79137421
MDG-a_39_n2000_m2	112283	1,68	10,0800991
MDG-a_40_n2000_m2	112709	1,30	9,87315178

Algoritmo Búsqueda Local			
Caso	Coste obtenido	Desv	Tiempo
GKD-c_11	19582,3494	0,02	559,5197678
GKD-c_12	19360,2239	0,00	578,0127048
GKD-c_13	19366,1494	0,00	564,1453266
GKD-c_14	19458,5647	0,00	649,4979858
GKD-c_15	19422,0506	0,00	602,0765305
GKD-c_16	19679,4583	0,00	560,9848499
GKD-c_17	19331,3884	0,00	611,9294167
GKD-c_18	19461,3946	0,00	588,4969234
GKD-c_19	19477,3258	0,00	601,8779278
GKD-c_20	19604,8436	0,00	582,8154087
MDG-b_1_	763792,0000	1,83	588,7620449
MDG-b_2_	759365,5700	2,64	555,1900864
MDG-b_3_	766015,4100	1,38	579,549551
MDG-b_4_	756112,5200	2,49	618,7927723
MDG-b_5_	767324,9100	1,07	642,6811218
MDG-b_6_	756305,4600	2,43	579,6532631
MDG-b_7_	756122,5300	2,72	639,7314072
MDG-b_8_	761155,5600	2,31	585,9601498
MDG-b_9_	764864,6500	1,28	566,9538975
MDG-b_10	761419,6400	1,75	579,9474716
MDG-a_31	112809	1,17	8137,60519
MDG-a_32	112982	0,97	7920,188665
MDG-a_33	112682	1,26	8241,734028
MDG-a_34	112514	1,48	8028,796196
MDG-a_35	113034	1,00	7956,383228
MDG-a_36	113277	0,85	8042,878389
MDG-a_37	112989	1,07	7830,316067
MDG-a_38	112894	1,30	7831,713438
MDG-a_39	112837	1,19	7795,49098
MDG-a_40	113122	0,94	7768,961906

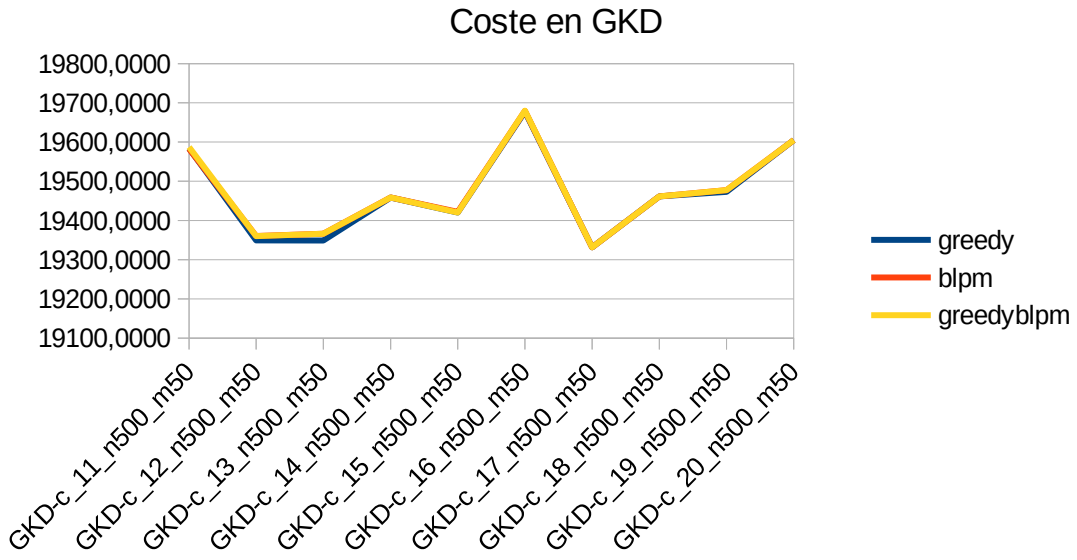
Media Desv: **1,37**
Media Tiempo: **3,76**

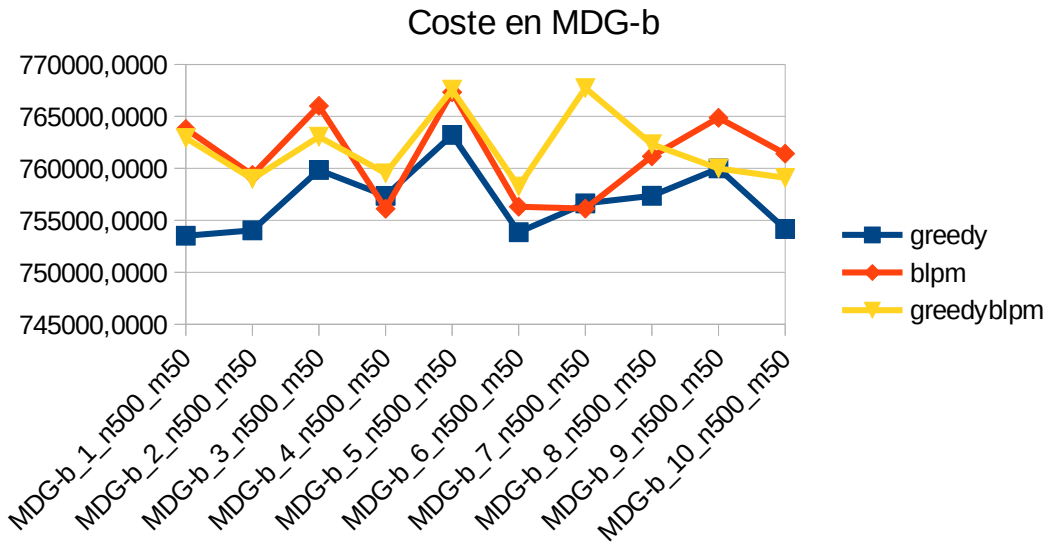
Media Desv: **1,04**
Media Tiempo: **3046,35**

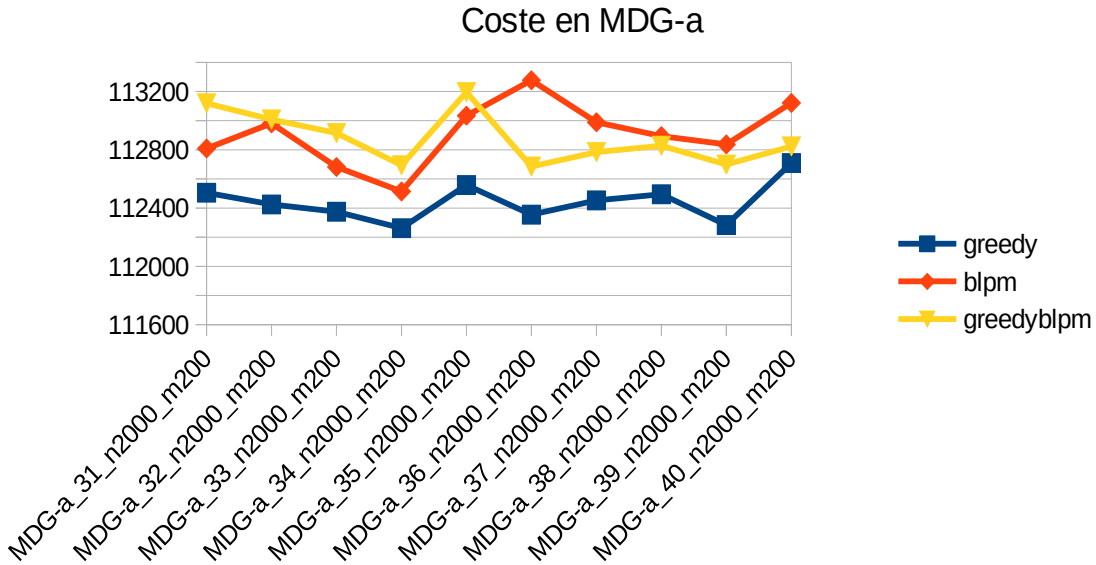
Algoritmo Greedy + BLPM			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
GKD-c_11	19587,12018	0,00	465,329885
GKD-c_12	19360,05736	0,00	386,380434
GKD-c_13	19366,14942	0,00	391,568899
GKD-c_14	19458,56466	0,00	394,316912
GKD-c_15	19420,26198	0,01	408,835888
GKD-c_16	19680,20477	0,00	402,001143
GKD-c_17	19331,38841	0,00	381,920099
GKD-c_18	19461,3946	0,00	382,081985
GKD-c_19	19477,32577	0,00	382,026434
GKD-c_20	19604,84356	0,00	388,484001
MDG-b_1_	762913,36	1,94	435,767174
MDG-b_2_	758993,56	2,69	416,211605
MDG-b_3_	763037,67	1,77	383,991718
MDG-b_4_	759517,52	2,05	382,856846
MDG-b_5_	767570,2	1,04	392,110825
MDG-b_6_	758239,22	2,18	415,146112
MDG-b_7_	767718,52	1,22	472,899675
MDG-b_8_	762320,95	2,16	418,308973
MDG-b_9_	759993,63	1,91	385,822535
MDG-b_10	759082,47	2,05	415,743351
MDG-a_31	113119	0,89	5441,26439
MDG-a_32	113008	0,95	5637,7542
MDG-a_33	112915	1,06	5590,48343
MDG-a_34	112694	1,32	5663,34534
MDG-a_35	113196	0,86	5702,91352
MDG-a_36	112685	1,37	5755,17058
MDG-a_37	112784	1,25	5506,39153
MDG-a_38	112828	1,36	5534,49798
MDG-a_39	112699	1,32	5662,83417
MDG-a_40	112823	1,20	5412,7028

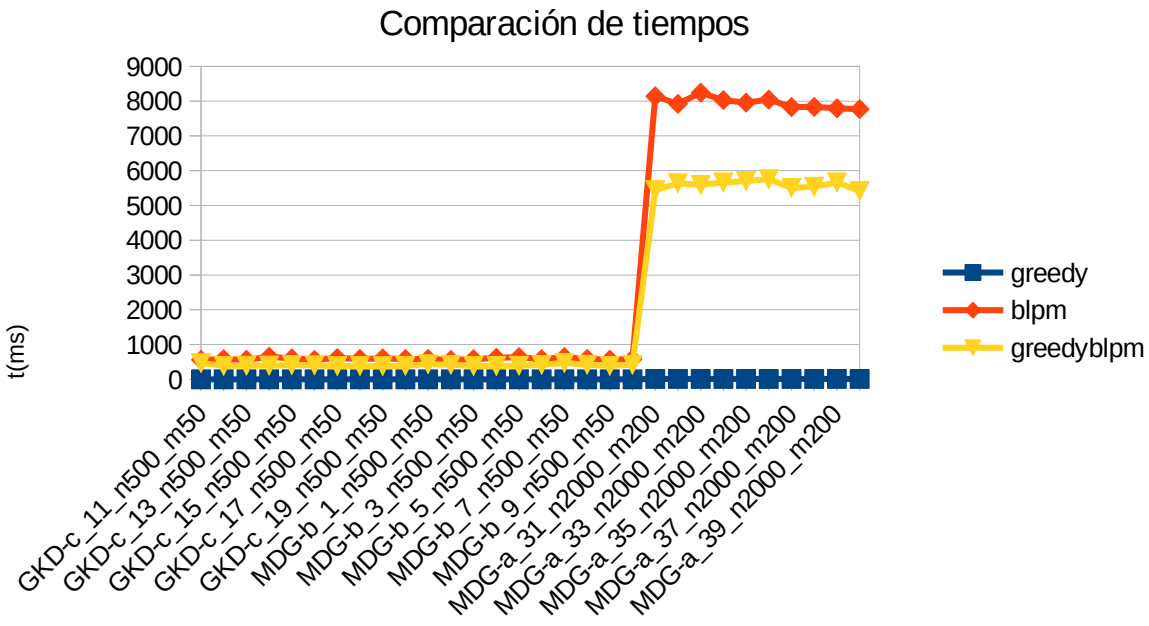
Media Desv: **1,02**

Media Tiempo: **2133,64**


Coste en GKD
19800,0000
19700,0000
19600,0000
19500,0000
19400,0000
19300,0000
19200,0000
19100,0000
GKD-c_11_n500_m50
GKD-c_12_n500_m50
GKD-c_13_n500_m50
GKD-c_14_n500_m50
GKD-c_15_n500_m50
GKD-c_16_n500_m50
GKD-c_17_n500_m50
GKD-c_18_n500_m50
GKD-c_19_n500_m50
GKD-c_20_n500_m50
greedy
blpm
greedyblpm


Coste en MDG-b
770000,0000
765000,0000
760000,0000
755000,0000
750000,0000
745000,0000
MDG-b_1_n500_m50
MDG-b_2_n500_m50
MDG-b_3_n500_m50
MDG-b_4_n500_m50
MDG-b_5_n500_m50
MDG-b_6_n500_m50
MDG-b_7_n500_m50
MDG-b_8_n500_m50
MDG-b_9_n500_m50
MDG-b_10_n500_m50
greedy
blpm
greedyblpm


Coste en MDG-a
113200
112800
112400
112000
111600
MDG-a_31_n2000_m200
MDG-a_32_n2000_m200
MDG-a_33_n2000_m200
MDG-a_34_n2000_m200
MDG-a_35_n2000_m200
MDG-a_36_n2000_m200
MDG-a_37_n2000_m200
MDG-a_38_n2000_m200
MDG-a_39_n2000_m200
MDG-a_40_n2000_m200
greedy
blpm
greedyblpm



El algoritmo Greedy es de media **2789,62 veces más rápido** que la búsqueda local. Es un algoritmo muy simple y rápido. Esto es totalmente lógico, ya que en cada iteración selecciona un elemento de la solución, y como es propio de los algoritmos voraces nunca revisa esta decisión, por lo tanto **solo itera m veces**.

La Búsqueda Local Primero el Mejor consigue optimizar más la función objetivo, sin embargo al iterar muchas más veces y explorar muchas soluciones vecinas, mínimo todas los vecinas a la solución inicial o 100.000 soluciones (lo que sea más pequeño), su tiempo de ejecución es considerablemente mayor.

Vamos a ver cuantas evaluaciones de soluciones ha realizado la BLMP:

n.º de evaluaciones en los Casos GKD del 11 al 20:

24293, 25778, 25179, 28802, 24854, 24088, 24674, 24206, 25166, 23744

n.º de evaluaciones en los casos MDG-a del 31 al 40:

100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000, 100000

n.º de evaluaciones en los casos MDG-b del 1 al 10:

25898, 25270, 26456, 27275, 28060, 25744, 28311, 25455, 25455, 25802

Analizando el n.º de evaluaciones realizadas por el algoritmo podemos observar que en los casos de n=500 el criterio de parada es no encontrar ninguna mejora en el vecindario, pero en los de n=2000 al aumentar el vecindario de cada solución enormemente **agota las iteraciones** y para forzosamente.

La variación probada Greedy+BLPM ha resultado ser un éxito. Es la opción con la que obtenemos **mejores resultados**, con una desviación media de 1,02, y es **1,42 veces más rápido** que la búsqueda local. Esto se consigue gracias a **converger más rápidamente** al partir de una buena solución, y se nota especialmente en los casos de tamaño más grande donde es más difícil converger.

Sin embargo la diferencia en la desviación media es tan pequeña (0.02) que no podemos tomar está medida como algo **significativo** y por lo tanto no podemos sacar conclusiones definitivas. La aleatoriedad del BLPM original podría haber dado mejores o iguales resultados.

Aunque los algoritmos que usan la Búsqueda Local mejoran las soluciones obtenidas respecto al greedy estos no siempre consiguen la solución óptima y siguen teniendo una desviación próxima a 1. Este comportamiento es el esperado ya que la Búsqueda Local cae en **óptimos locales** del espacio de búsqueda.

Como conclusión del análisis el algoritmo más adecuado a usar **dependería del tamaño del problema** y el **tiempo disponible**. Si tenemos un tamaño muy grande y unas restricciones de tiempo muy estrictas podríamos conformarnos con la solución del Greedy. Si tenemos mucho tiempo y queremos la mejor solución lo mejor sería lanzar el algoritmo de BLPM varias veces con distintas semillas y probar también con el greedy+BLPM. Si tenemos un tiempo limitado pero queremos una solución de calidad la mejor opción sería el algoritmo greedy+BLPM.