Metaheurísticas

Alberto Estepa Fernández – 31014191W
albertoestep@correo.ugr.es
Gr. Prácticas MH2 - Jueves 17:30h
Estudiante del Doble Grado de Ingeniería Informática y Matemáticas

Práctica 1.a: Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema de la Máxima Diversidad

2019/2020

UNIVERSIDAD
DE GRANADA

13 de marzo de 2020

Índice

Descripción del problema

El Problema de la Máxima Diversidad (Maximun Diversity Problem en inglés, MDP) es un problema de optimización combinatoria consistente en seleccionar un subconjunto M de m elementos (|M|=m) de un conjunto inicial S de n elementos (obviamente, n>m) de forma que se maximice la diversidad entre los elementos escogidos.

Además de los n elementos (e_i, i=1,...,n) y el número de elementos a seleccionar m, se dispone de una matriz D=(d_{ij}) de dimensión nxn que contiene las distancias entre ellos.

En este caso la diversidad se calcula como la suma de las distancias entre cada par de elementos seleccionados.

Matemáticamente el problema consistiría en maximizar la función:

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

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

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

donde x es el vector solución al problema.

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_11_n500_m50 a GKD-c_20_n500_m50), 10 del grupo MDG con distancias reales en [0,1000] , n=500 y m=50 (MDG-b_1_n500_m50 a MDG-b_10_n500_m50); y 10 del grupo MDG con distancias enteras en {0,10}, n=2000 y m=200 (MDG-a_31_n2000_m200 a MDG-a_40_n2000_m200).

Descripción de la aplicación de los algoritmos empleados

En este apartado describiremos las consideraciones comunes a los distintos algoritmos. Incluimos la representación de las soluciones y la función objetivo.

Se ha usado C++ para la realización del código de la práctica.

A) Representación de las soluciones

El esquema de representación de una solución es el siguiente:

```
struct solucion{
        vector<int> v;
        double coste;
};
```

Aquí el vector v contiene los índices de los puntos que forman la solución (es un vector de tamaño m). Los índices son valores no repetidos entre 1 y n.

Los datos de donde hemos sacado las distancias se han almacenado en una matriz simétrica de tamaño nxn.

B) Función objetivo

Esta primera función calcula, dado un conjunto de elementos y un elemento concreto, la distancia acumulada entre el elemento y el cada uno de los elementos del conjunto.

```
Parámetros: elemento, conjunto y matriz de distancias
Resultado: suma acumulada de distancias del elemento al conjunto.
distanciaAUnConjunto{
        suma = 0
        Para cada i ∈ conjunto hacer:
                suma = matriz[elemento][i]
        Fin para cada
        Devolver suma
}
```

La siguiente función calcula, dado un conjunto de elementos que forma la solución, la distancia entre los elementos del conjunto, es decir, el coste de la solución.

Parámetros: conjunto_solución y matriz de distancias
Resultado: coste de la solución.
costeSolucion*{*
    coste = 0
    Para cada i ∈ conjunto hacer:
        coste = coste + distancia acumulada de i al conjunto_solución
    Fin para cada
    Devolver coste / 2
}

Descripción de la estructura de los métodos

Disponemos de dos algoritmos para la búsqueda. En este apartado describiremos la estructura de dichos métodos y las operaciones relevantes.

A) Greedy

La idea del método es añadir secuencialmente el elemento no seleccionado que más diversidad aporte con respecto a los ya seleccionados. Para ello, partiremos del elemento más alejado del resto en el conjunto completo de elementos. En los m-1 pasos siguientes se va escogiendo el elemento más lejano a los elementos seleccionados hasta el momento. Cuando hablamos de "más lejano respecto a los demás" nos referimos a aquel elemento cuya suma acumulada de las distancias a los demás elementos sea la mayor.

Para éste problema hemos creado la función masLejano(·) que dados dos conjuntos de elementos, calcula el elemento del segundo conjunto (del conjunto de los elementos todavía no seleccionados para la solución) que está más alejado de los elementos del primer conjunto (del conjunto de los elementos ya seleccionados para la solución).

Parámetros: seleccionados, no_seleccionados y *matriz de distancias*
Resultado: elemento de no_seleccionados más lejano a seleccionados.
m***asLejano****{*
    distancia_máxima = 0
    PARA CADA elemento ∈ no_seleccionados HACER:
        distancia_actual = distanciaAUnConjunto(elemento, seleccionados, m)
        SI distancia_actual > distancia_máxima HACER:
            distancia_máxima = distancia_actual
            solución = elemento
        FIN SI
    FIN PARA CADA
    DEVOLVER solución
}

Ya podemos implementar el algoritmo Greedy a partir de los métodos y la explicación anterior:

Parámetros: matriz de distancias, numero_elementos_solución
Resultado: solución_calculada y coste de la solución.
greedy*{*
    no_seleccionados = {0, …, n}
    seleccionados = { masLejano(no_seleccionados, no_seleccionados, m)}
    MIENTRAS numero_elementos_seleccionados < numero_elementos_solución HACER:
        elemento_mas_lejano = masLejano(seleccionados, no_seleccionados, m);
        seleccionados = seleccionados ∪ elemento_mas_lejano
        no_seleccionados = no_seleccionados – {elemento_mas_lejano}
    FIN MIENTRAS
    coste = costeSolucion(seleccionados, m)
    DEVOLVER coste}

B) Búsqueda local

Para éste método, partimos de una solución aleatoria válida y la iremos mejorando intercambiando un elemento por otro de su vecindario que mejore la solución si existe. Usaremos la estrategia del primero mejor, que consiste en cada vez que encontremos un elemento que mejore a otro en la solución dada, realizaremos el intercambio y pasaremos a la siguiente iteración. Pararemos el método cuando lleguemos a un límite de iteraciones en las que no se mejore la solución (100.000 iteraciones en nuestro caso).

Parámetros: matriz de distancias, numero_elementos_solución
Resultado: solución_calculada y coste de la solución.
busquedaLocal*{*
    solución = solucionAleatoria(·)
    MIENTRAS sigamos_mejorando_solución HACER:
        sigamos_mejorando_solución = exploracionVecindario(solución, matriz_distancias)
    FIN MIENTRAS
    DEVOLVER coste
}

Hacemos varias aclaraciones de este método: Por un lado tenemos la función solucionAleatoria(·) que como su propio nombre indica calcula una solución del problema de forma aleatoria. Incluimos a continuación el pseudocódigo de dicho método

---

Parámetros: solución_a_calcular, numero_elementos_solución y matriz_de_distancias.
Resultado: solución_calculada.
solucionAleatoria*{*
    seleccionados = 0
    MIENTRAS seleccionados < numero_elementos_solución HACER:
        índice_aleatorio = generarNumeroAleatorio(·)
        SI índice_aleatorio ∉ solución HACER
            solución = solución ∪ {índice_aleatorio}
            seleccionados = seleccionados + 1
        FIN SI
    FIN MIENTRAS
    coste = costeSolucion(·)
}

Por otro lado, el método más importante de la búsqueda local es exploracionVecindario(·) que explora el vecindario de las soluciones para ver si algún vecino mejora la solución anterior. Para ello hemos usado la factorización de la búsqueda, es decir, para ver si una solución es mejor que otra al realizar el intercambio de un elemento por otro, estudiaremos si dicho intercambio mejora la solución y no calcularemos el coste de la nueva solución para calcularla con la actual. Esto simplifica significativamente el cómputo.

Para ello lo primero que hemos hecho es ordenar los elementos de la solución de menor a mayor por cantidad de contribución que aportan al coste de la solución. Esto tiene sentido porque será más útil estudiar aquellos elementos que tengan menor contribución en la solución primero, pues la intuición indica que es más probable que encontremos elementos vecinos cuya contribución a la solución sea mayor. Para implementar ésto hemos usado la función ordenaSolucionPorContribucion(·) que explicaremos más adelante.

Como indica la práctica, dejaremos de explorar el vecindario cuando no encontremos mejora en la solución para unas ciertas iteraciones en la ejecución (en nuestro caso este número será 100.000 iteraciones).

Así para cada elemento de la solución buscaremos un elemento vecino aleatorio que mejore al anterior y si lo encontramos realizamos el intercambio y volvemos a ejecutar la exploración del vecindario para esta solución ya actualizada. A continuación incluimos el pseudocódigo:

```
Parámetros: solución_a_mejorar y matriz_de_distancias.
Resultado: booleano_que_indica_si_se_ha_mejorado.
exploracionVecindario{
        solución = ordenaSolucionPorContribucion(·)
        limite_iteraciones = 100.000 / tamaño_solución
        i = 0
        MIENTRAS i < numero_elementos_solución HACER:
                elemento_actual = solución[i]
                antigua_contribución =  distanciaAUnConjunto(elemento_actual, solución)
                k = 0
                índice_aleatorio = generarNumeroAleatorio(·)
                MIENTRAS k < limite_iteraciones HACER:
                        SI índice_aleatorio ∉ solución HACER:
                                nueva_contribución  =  distanciaAUnConjunto(índice_aleatorio,  /
                                        / solución) – matriz[índice_aleatorio][elemento_actual]
                                SI nueva_contribución > antigua_contribución HACER:
                                        solución[i] =  índice_aleatorio
                                        coste = coste + nueva_contribución - antigua_contribución;
                                        DEVOLVER seguir_mejorando
                                FIN SI
                                k = k + 1
                        FIN SI
                        índice_aleatorio = generarNumeroAleatorio(·)
                FIN MIENTRAS
                i = i + 1
        FIN MIENTRAS
        DEVOLVER parar_algoritmo
}
```

Por último, falta explicar como hemos implementado ordenaSolucionPorContribucion(·). Para ello hemos sobrecargado el operador < para estructuras pair<int, double> donde el primer valor indica el elemento usado y el segundo la contribución que hace al coste de la solución.

Así usaremos la función sort(·) de vectores para ésta función y obtendremos la solución ordenada.

Procedimiento considerado para desarrollar la práctica

Todo el código se ha implementado en C++. El procedimiento para el desarrollo de la práctica ha seguido las explicaciones dadas en la memoria. El código ha sido implementado por mí siguiendo el Seminario 2 de la asignatura, el guión de la práctica y las explicaciones dadas en clase.

Está estructurado en una serie de carpetas: bin (donde se encontraran los ejecutables), entrada (donde se encuentran las ficheros de datos proporcionados), resultados (donde se encuentran las tablas pedidas con los resultados de las ejecuciones), src (donde se encuentran los dos ficheros creados para la práctica: greedy.cpp donde se incluye el algoritmo greedy y busquedaLocal.cpp donde se incluye el algoritmo de búsqueda local implementado) y un makefile.

El makefile realiza toda la ejecución. Dispone de varias reglas entre las que destacamos:

- make clean: limpia los ejecutables creados anteriormente
- make: ejecuta los dos algoritmos con algunos de los problemas de ejemplo.
- make Resultados: realiza las 60 ejecuciones necesarias para obtener los datos

El fichero makefile ha ejecutado con opciones de optimización -O2.

El fichero busquedaLocal.cpp debe ejecutarse introduciendo un valor de semilla. Este valor será introducido en el Makefile en mi caso y se puede cambiar en la variable del makefile SEMILLA.

Los resultados obtenidos se deben a que el PC usado para dichas ejecuciones consta de:

- Sistema operativo Ubuntu 18.10
- Memoria de 7885 M
- Procesador Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz

Experimentos y análisis de resultados

A continuación plasmamos los datos recogidos por las ejecuciones propuestas para los dos algoritmos:

A) **Greedy**

Algoritmo Greedy				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo
GKD-c_11_n500_m50	18095,5	19587,12891	7,62	0,004699
GKD-c_12_n500_m50	17356,5	19360,23633	10,35	0,004527
GKD-c_13_n500_m50	17677,1	19366,69922	8,72	0,004528
GKD-c_14_n500_m50	17529,1	19458,56641	9,92	0,004685
GKD-c_15_n500_m50	17752,6	19422,15039	8,60	0,003669
GKD-c_16_n500_m50	17978,2	19680,20898	8,65	0,003698
GKD-c_17_n500_m50	17903,5	19331,38867	7,39	0,004701
GKD-c_18_n500_m50	17713,5	19461,39453	8,98	0,005183
GKD-c_19_n500_m50	17704,9	19477,32813	9,10	0,003710
GKD-c_20_n500_m50	17788,8	19604,84375	9,26	0,003841
MDG-b_1_n500_m50	754081,0	778030,625	3,08	0,004553
MDG-b_2_n500_m50	761437,0	779963,6875	2,38	0,003946
MDG-b_3_n500_m50	751115,0	776768,4375	3,30	0,003633
MDG-b_4_n500_m50	760257,0	775394,625	1,95	0,008008
MDG-b_5_n500_m50	753668,0	775611,0625	2,83	0,007776
MDG-b_6_n500_m50	753581,0	775153,6875	2,78	0,007361
MDG-b_7_n500_m50	755966,0	777232,875	2,74	0,007616
MDG-b_8_n500_m50	755936,0	779168,75	2,98	0,007501
MDG-b_9_n500_m50	752100,0	774802,1875	2,93	0,008186
MDG-b_10_n500_m50	745121,0	774961,3125	3,85	0,007422
MDG-a_31_n2000_m200	112451,0	114139	1,48	0,419913
MDG-a_32_n2000_m200	112390,0	114092	1,49	0,551197
MDG-a_33_n2000_m200	112338,0	114124	1,56	0,432297
MDG-a_34_n2000_m200	112694,0	114203	1,32	0,567860
MDG-a_35_n2000_m200	112288,0	114180	1,66	0,590339
MDG-a_36_n2000_m200	112296,0	114252	1,71	0,533663
MDG-a_37_n2000_m200	111986,0	114213	1,95	0,481793
MDG-a_38_n2000_m200	112298,0	114378	1,82	0,525398
MDG-a_39_n2000_m200	112326,0	114201	1,64	0,540270
MDG-a_40_n2000_m200	112833,0	114191	1,19	0,533073

Calculando la media de las desviaciones y la media de los tiempos obtenemos:

Número Casos	30
Media Desv:	4,44083647313
Media Tiempo:	0,1761682

B) **Búsqueda Local**

Algoritmo Búsqueda Local				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo
GKD-c_11_n500_m50	18280,0	19587,12891	6,67	0,500024
GKD-c_12_n500_m50	17755,6	19360,23633	8,29	0,575608
GKD-c_13_n500_m50	17935,8	19366,69922	7,39	1,275980
GKD-c_14_n500_m50	17902,2	19458,56641	8,00	1,024850
GKD-c_15_n500_m50	17800,6	19422,15039	8,35	0,534940
GKD-c_16_n500_m50	18175,5	19680,20898	7,65	0,520452
GKD-c_17_n500_m50	17972,5	19331,38867	7,03	1,661910
GKD-c_18_n500_m50	17869,2	19461,39453	8,18	1,450120
GKD-c_19_n500_m50	17974,0	19477,32813	7,72	0,831387
GKD-c_20_n500_m50	17887,9	19604,84375	8,76	1,489660
MDG-b_1_n500_m50	770730,0	778030,625	0,94	0,974748
MDG-b_2_n500_m50	758308,0	779963,6875	2,78	0,744570
MDG-b_3_n500_m50	766037,0	776768,4375	1,38	1,039270
MDG-b_4_n500_m50	764054,0	775394,625	1,46	1,398590
MDG-b_5_n500_m50	756103,0	775611,0625	2,52	0,484578
MDG-b_6_n500_m50	761191,0	775153,6875	1,80	0,683489
MDG-b_7_n500_m50	766263,0	777232,875	1,41	0,959520
MDG-b_8_n500_m50	770877,0	779168,75	1,06	1,053690
MDG-b_9_n500_m50	764975,0	774802,1875	1,27	1,287680
MDG-b_10_n500_m50	752744,0	774961,3125	2,87	1,034490
MDG-a_31_n2000_m200	113640,0	114139	0,44	27,857500
MDG-a_32_n2000_m200	113234,0	114092	0,75	36,456600
MDG-a_33_n2000_m200	113092,0	114124	0,90	31,166400
MDG-a_34_n2000_m200	113422,0	114203	0,68	35,530300
MDG-a_35_n2000_m200	112753,0	114180	1,25	37,656500
MDG-a_36_n2000_m200	113310,0	114252	0,82	31,575100
MDG-a_37_n2000_m200	113501,0	114213	0,62	25,039000
MDG-a_38_n2000_m200	113302,0	114378	0,94	40,545600
MDG-a_39_n2000_m200	112674,0	114201	1,34	29,100800
MDG-a_40_n2000_m200	112959,0	114191	1,08	27,268400

Calculando la media de las desviaciones y la media de los tiempos obtenemos:

Número Casos	30
Media Desv:	3,4782723
Media Tiempo:	11,3907252

Conclusiones:

Lo primero que podemos observar, dados los resultados anteriores es que ambos métodos llegan a soluciones bastante notables. Más concretamente para el algoritmo greedy los resultados obtenidos son buenos teniendo en cuenta el poco coste en cuanto a complejidad computacional y sencillo código. Nos permiten dar una aproximación del óptimo. Para el algoritmo de búsqueda local, los resultados son también bastante buenos y algo mejores que los obtenidos por el algoritmo greedy (plasmaremos una comparativa entre ambos métodos a continuación).

Por otro lado, se aprecia un coste en tiempo bastante grande cuando aumentamos el tamaño de los datos, como podemos apreciar en los problemas MDG-a_x_n2000_m200 con respecto a los anteriores. Este coste es bastante significativo en la búsqueda local pues pasamos de ejecutar el algoritmo en tiempos cercanos al segundo a unos treinta segundos con los problemas de gran tamaño (un aumento de 30 veces más). Sin embargo, aunque los tiempos en el greedy son significativamente menores, el aumento de los problemas de mayor tamaño respecto a los problemas de menor tamaño es de aproximadamente 50 veces más.

A pesar de la cercanía de los resultados al óptimo, parece sin embargo que dicho óptimo es dificil de alcanzar por nuestros algoritmos y que siempre nos quedamos con un déficit alrededor de 2000 ó 4000 unidades de coste en todos los casos, sin distinciones notables entre problemas cuyo coste óptimo es mayor o menor. Así pues el parámetro de la desviación de los datos no parece tan significativo como quisiéramos.

Por otro lado, iniciamos una comparativa entre los dos algoritmos:

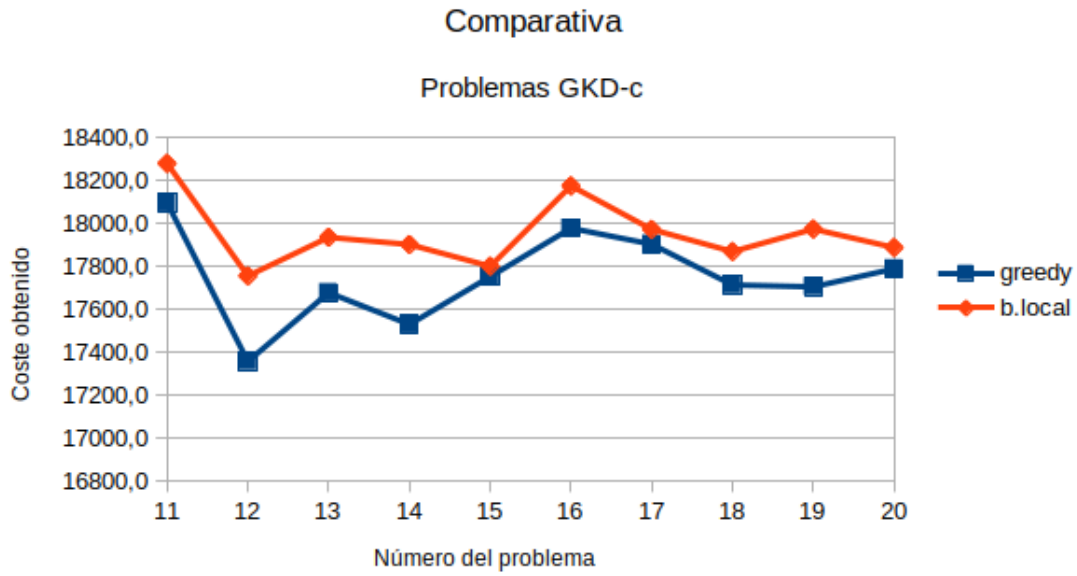
	Algoritmo	
	Greedy	Búsqueda local
Media Desv:	4,44083647313	3,478272299743
Media Tiempo:	0,1761682	11,3907252

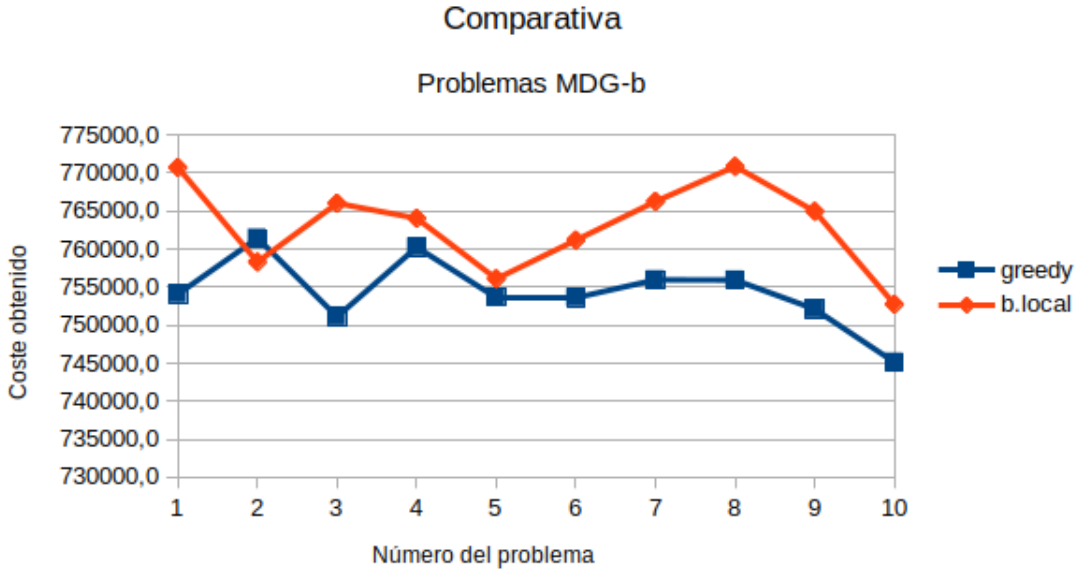
Por lo pronto, analizamos la tabla conjunta de tiempos y desviaciones medios entre los dos algoritmos: como preveíamos, el método de búsqueda local es mejor en resultado de coste, pues la desviación media de los datos al óptimo es un punto mejor que la obtenida con el método greedy. Dicha mejora se costea por otro lado con el tiempo de ejecución entre ambos algoritmos, que, obviamente, el algoritmo de búsqueda local es significativamente peor en este aspecto.

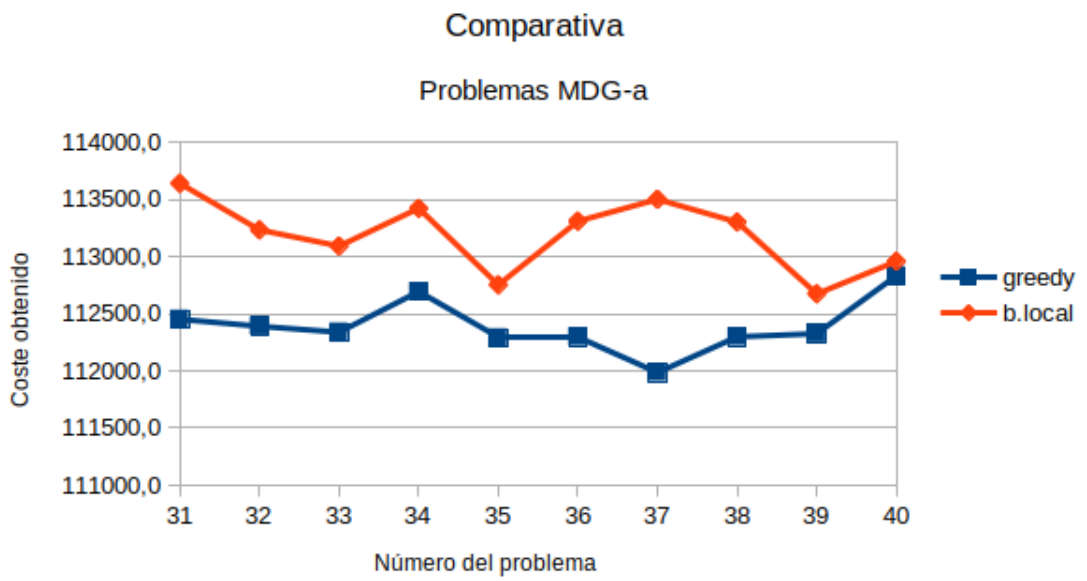
Desgranando dichos resultados, podemos analizar uno a uno los resultados obtenidos para cada problema con ambos algoritmos. Es la tabla que justamente tenemos debajo. Vemos que salvo casos concretos, en general el algoritmo de búsqueda local mejora los resultados obtenidos por el algoritmo greedy, sin embargo cabe destacar que no son para nada significativos estas mejoras y en algunos casos específicos el método greedy es mejor que el algoritmo de búsqueda local. Éstos ejemplos son una clara muestra de como un algoritmo puede ser más efectivo frente a un caso de estudio particular y no serlo en otro.

	Algoritmo	
	Greedy	Búsqueda local
Caso	Coste obtenido	
GKD-c_11_n500_m50	18095,5	18280,0
GKD-c_12_n500_m50	17356,5	17755,6
GKD-c_13_n500_m50	17677,1	17935,8
GKD-c_14_n500_m50	17529,1	17902,2
GKD-c_15_n500_m50	17752,6	17800,6
GKD-c_16_n500_m50	17978,2	18175,5
GKD-c_17_n500_m50	17903,5	17972,5
GKD-c_18_n500_m50	17713,5	17869,2
GKD-c_19_n500_m50	17704,9	17974,0
GKD-c_20_n500_m50	17788,8	17887,9
MDG-b_1_n500_m50	754081,0	770730,0
MDG-b_2_n500_m50	761437,0	758308,0
MDG-b_3_n500_m50	751115,0	766037,0
MDG-b_4_n500_m50	760257,0	764054,0
MDG-b_5_n500_m50	753668,0	756103,0
MDG-b_6_n500_m50	753581,0	761191,0
MDG-b_7_n500_m50	755966,0	766263,0
MDG-b_8_n500_m50	755936,0	770877,0
MDG-b_9_n500_m50	752100,0	764975,0
MDG-b_10_n500_m50	745121,0	752744,0
MDG-a_31_n2000_m200	112451,0	113640,0
MDG-a_32_n2000_m200	112390,0	113234,0
MDG-a_33_n2000_m200	112338,0	113092,0
MDG-a_34_n2000_m200	112694,0	113422,0
MDG-a_35_n2000_m200	112288,0	112753,0
MDG-a_36_n2000_m200	112296,0	113310,0
MDG-a_37_n2000_m200	111986,0	113501,0
MDG-a_38_n2000_m200	112298,0	113302,0
MDG-a_39_n2000_m200	112326,0	112674,0
MDG-a_40_n2000_m200	112833,0	112959,0

Para visualizarlo correctamente, hemos realizado gráficas comparativas para cada una de las tres clases de problemas. Adjuntamos dichas gráficas a continuación y en ellas podemos apreciar lo discutido anteriormente:


Comparativa
Problemas GKD-c
Coste obtenido
18400,0
18200,0
18000,0
17800,0
17600,0
17400,0
17200,0
17000,0
16800,0
11
12
13
14
15
16
17
18
19
20
Número del problema
greedy
b.local


Comparativa
Problemas MDG-b
Coste obtenido
775000,0
770000,0
765000,0
760000,0
755000,0
750000,0
745000,0
740000,0
735000,0
730000,0
1
2
3
4
5
6
7
8
9
10
Número del problema
greedy
b.local


Comparativa
Problemas MDG-a
Coste obtenido
114000,0
113500,0
113000,0
112500,0
112000,0
111500,0
111000,0
31
32
33
34
35
36
37
38
39
40
Número del problema
greedy
b.local

Referencias bibliográficas

No he utilizado libros pero si he encontrado algunos links útiles para ciertas funciones del proyecto. A continuación los muestro:

- Para todo el tema de la implementación usé la documentación de la stl de C++ http://www.cplusplus.com/reference/stl/

- Para el tema de la generación de números aleatorios y fijar la semilla, me basé en implementaciones vistas en https://stackoverflow.com

- Busqué información sobre el tema a tratar en varios lugares como http://luna.inf.um.es/grupo_investigacion/PFCs_y_TMs/2015memoriaPFC_MiguelAngelFranco.pdf ó https://ccc.inaoep.mx/~emorales/Cursos/Busqueda/blocal.pdf

- PRADO y página web de la asignatura.