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 3.a: Búsquedas por Trayectorias para el Problema de la Máxima Diversidad

2019/2020

UNIVERSIDAD DE GRANADA

25 de mayo 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**

Hemos usado la representación binaria de la práctica anterior para la implementación de los algoritmos de ésta práctica. Ésto lo hemos considerado justificado debido a que podíamos reutilizar código de dicha práctica de forma sencilla y no supone un gran cambio a otra representación en éste caso. El esquema resultante es el siguiente:

```
struct solucion{
        vector<bool> vector;
        double diversidad;
        bool evaluada;
};
```

Aquí el vector 'vector' contiene un 1 en los índices de los puntos que forman la solución y un 0 en los demás (ahora es un vector de tamaño n en vez de tamaño m como en la práctica 1). Los datos de donde hemos sacado las distancias se han guardado en una matriz simétrica de tamaño nxn.

En los algoritmos donde se usa hace uso del algoritmo de búsqueda local (BMB e ILS) necesitaremos la representación dada por índices enteros, así que sólo en estos casos y de forma momentánea usaremos la siguiente representación (la adaptación entre representaciones es la misma que la implementada en la práctica 2):

```
struct solucion_ints{
        vector<int> vector;
        double diversidad;
};
```

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, la diversidad de la solución.

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

Aún así se han incorporado la función de evaluación de una solución para hacerlo más entendible, y permitir el evitar hacer cálculos innecesarios modificando la bandera ‘evaluada’ cuando sea necesario.

Parámetros: solución (s) y matriz de distancias
Resultado: calcula la diversidad entre los elemento de la solución y pone el flag evaluada a true.
evaluarSolucion*{*
    diversidad de s = costeSolucion()
    flag evaluada = true
    Devolver diversidad de s
}

C) Generación de soluciones aleatorias

En todos los algoritmos siguientes generaremos en primera instancia una solución aleatoria para lo cual hemos implementado la función solucionAleatoria(·). Incluimos a continuación el pseudocódigo de dicho método:

```
Parámetros: solución_a_calcular (s), numero_elementos_solución y matriz_de_distancias.
Resultado: solución_calculada.
solucionAleatoria{
        seleccionados = 0
        s.evaluada = false
        s.diversidad = 0
        Inicializar s.vector a false (tantos como elementos haya en el dominio)
        MIENTRAS seleccionados < numero_elementos_solución HACER:
                índice_aleatorio = generarNumeroAleatorio(·)
                SI s.vector[índice_aleatorio] = false HACER:
                        s.vector[índice_aleatorio] = true
                        seleccionados = seleccionados + 1
                FIN SI
        FIN MIENTRAS
}
```

Descripción de la estructura de los métodos

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

A) Enfriamiento Simulado (ES)

El enfriamiento simulado como ya sabemos es un método de búsqueda por entornos caracterizado por un criterio de aceptación de soluciones vecinas que se adapta a lo largo de su ejecución. Hace uso de una variable llamada temperatura actual, t, cuyo valor determina en qué medida pueden ser aceptadas soluciones vecinas peores que la actual. La variable temperatura acutal se va reduciendo cada iteración mediante un mecanismo de enfriamiento de la temperatura, hasta alcanzar una temperatura final, t_f. En cada iteración se genera un número concreto de vecinos, y cada vez que se genera un vecino, se aplica el criterio de aceptación para ver si sustituye a la solución actual. Si la solución vecina es mejor que la actual, se acepta automáticamente, tal como se haría en la búsqueda local clásica. En cambio, si es peor, aún existe la probabilidad de que el vecino sustituya a la solución actual. Esto permite al algoritmo salir de óptimos locales.

Parámetros: n.º de elementos a elegir (n), matriz de distancias (m) y n.º máx. de evaluaciones
Resultado: solución calculada y diversidad de la solución.
enfriamientoSimulado*{*
Inicializamos evaluaciones = 0, máx. vecinos, éxitos = 1, máx éxitos, mejor diversidad = 0, temperatura inicial, temperatura final (t_f) y n.º de enfriamientos.
solución actual = solucionAleatoria(·)
evaluarSolucion(solución actual)
solución guardada = solución actual
mejor_diversidad = diversidad de la solución actual
Inicializamos temperatura inicial y beta
temperatura actual (t) = temperatura inicial
MIENTRAS éxitos > 0 y evaluaciones < n.º máx. de evaluaciones y t < t_f HACER:
éxitos = 0
PARA CADA i ∈ [0, max_vecinos – 1] MIENTRAS QUE éxitos < máx. éxitos HACER:
solución actual = mutarSolucion(solución actual)
evaluaciones = evaluaciones + 1
delta = diversidad solución actual – diversidad solución guardada
n.º aleatorio ∈ [0,1]
SI delta > 0 y exp(delta / t) HACER:
solución guardada = solución actual
éxitos = éxitos + 1
SI diversidad solución actual > mejor diversidad HACER:
mejor diversidad = diversidad solución actual
FIN SI
SINO HACER:
solución actual = solución guardada
FIN SI
FIN PARA CADA
*temperatura actual = temperatura_actual / (1 + beta * temperatura_actual)*
FIN MIENTRAS
DEVOLVER solución actual
}

Hacemos varias aclaraciones de este método: Por un lado usamos la función siguiente para calcular los vecinos:

Parámetros: solución a mutar (s), n.º de evaluaciones de la función objetivo, y matriz de distancias.
Resultado: Solución mutada y n.º de evaluaciones.
mutarSolucion{
HACER: n.º aleatorio1= aleatorio entre 0 y s.vector.size() MIENTRAS n.º aletorio1 = false
HACER: n.º aleatorio2= aleatorio entre 0 y s.vector.size() MIENTRAS n.º aletorio2 = true
s.vector[n.º aleatorio1] = false
s.vector[n.º aleatorio2] = true
SI s.evaluada = true HACER:
antigua_diversidad = distanciaAUnConjunto(n.º aleatorio1, s.vector, m) –
matriz_distancias[n.º aleatorio1][n.º aleatorio2]
nueva_diversidad = distanciaAUnConjunto(n.º aleatorio2, s.vector, m)
s.coste = s.coste + nueva_diversidad – antigua_diversidad
n.º evaluaciones = n.º evaluaciones + 1
FIN SI
DEVOLVER s y n.º de evaluaciones
}

Por otro lado cabe destacar que hemos obtenido resultados notables con las inicializaciones propuestas en el guión, salvo pequeñas modificaciones:

- máx. vecinos = tamaño del caso del problema (en el guión aconsejaban 10 veces dicho tamaño).
- máx. éxitos = 0.1 * máx. vecinos
- temperatura final = 10^{-3}
- temperatura inicial = diversidad de la solución inicial * 0.3 / (- log(0.3))
- n.º máx de evaluaciones = 100.000
- n.º de enfriamientos = 100.000 / máx. vecinos
- enfriamiento de Cauchy modificado con beta = (temperatura inicial – temperatura final) / (n.º de enfriamientos * temperatura inicial * temperatura final)

B) Búsqueda Multiarranque Básica (BMB)

Para éste método simplemente generamos un determinado número de soluciones aleatorias iniciales (10 en concreto) y optimizamos cada una de ellas con el algoritmo de búsqueda local de la Práctica 1.

<u>Parámetros:</u> n.º de elementos a elegir (n), matriz de distancias (m) y n.º máx. de evaluaciones
<u>Resultado:</u> solución calculada y diversidad de la solución.
enfriamientoSimulado*{*
    Inicializamos evaluaciones = 0, iteraciones, mejor diversidad = 0, solución (s)
    PARA CADA i ∈ [0, iteraciones – 1] HACER:
        solución actual = solucionAleatoria(·)
        evaluarSolucion(solución actual)
        evaluaciones = 1 + evaluaciones + busquedaLocal(s, n.º máx. de evaluaciones)
        SI diversidad de s > mejor diversidad HACER:
            mejor diversidad = diversidad de s
        FIN SI
    FIN PARA CADA
    DEVOLVER s y diversidad de s
}

Recordemos que la función busquedaLocal de la Práctica 1 devolvía el número de evaluaciones que realizaba (y la solución actualizada pasada como parámetro).

Por otro lado hemos inicializado las iteraciones a 10 y el n.º máx. de evaluaciones a 10.000 como se sugería en el guión de la práctica, obteniendo resultados bastante notables.

C) Búsqueda Local Reiterada (ILS)

Para éste método, generaremos una solución inicial aleatoria y aplicaremos el algoritmo de búsqueda local sobre ella. Una vez obtenida la solución optimizada, se estudiaremos si es mejor que la mejor solución encontrada hasta el momento y realizaremos una mutación sobre la mejor de estas dos, volviendo a aplicar el algoritmo de búsqueda local sobre esta solución mutada. Este proceso se repetirá un determinado número de veces, devolviéndose la mejor solución encontrada en toda la ejecución. Por tanto, se seguirá el criterio del mejor como criterio de aceptación de la ILS.

Parámetros: n.º de elementos a elegir (n), matriz de distancias (m) y n.º máx. de evaluaciones
Resultado: solución calculada y diversidad de la solución.

```
ILS{
    Inicializamos evaluaciones, iteraciones y mejor diversidad = 0.
    solución actual = solucionAleatoria(·)
    evaluarSolucion(solución actual)
    solución guardada =  solución actual
    mejor diversidad = diversidad solución actual
    evaluaciones = 1 + busquedaLocal(solución actual, n.º máx. de evaluaciones)
    SI diversidad solución guardada > diversidad solución actual HACER:
        solución actual =  solución guardada
    SINO HACER:
        solución guardada =  solución actual
        SI diversidad solución actual > mejor diversidad HACER:
            mejor diversidad =  diversidad solución actual
        FIN SI
    FIN SI
    PARA CADA i ∈ [1, iteraciones – 1] HACER:
        perturbacionBrusca(solución actual)
        evaluaciones = 1 + evaluaciones + busquedaLocal(s. actual, n.º máx. de eval.)
        SI diversidad solución guardada > diversidad solución actual HACER:
            solución actual =  solución guardada
        SINO HACER:
            solución guardada =  solución actual
            SI diversidad solución actual > mejor diversidad HACER:
                mejor diversidad =  diversidad solución actual
            FIN SI
        FIN SI
    FIN PARA CADA
    DEVOLVER solución guardada y mejor diversidad
}
```

Recordemos que la función busquedaLocal de la Práctica 1 devolvía el número de evaluaciones que realizaba (y la solución actualizada pasada como parámetro). Por otro lado hemos inicializado las iteraciones a 10 y el n.º máx. de evaluaciones a 10.000 como se sugería en el guión de la práctica, obteniendo resultados bastante notables.

Cabe destacar la función de obtención de vecinos:

<u>Parámetros:</u> solución a perturbar, n.º de elementos a elegir en una solución
<u>Resultado:</u> solución calculada.
perturbacionBrusca*{*
*n.º de mutaciones = 0.1 * n.º de elementos a elegir en una solución*
PARA CADA i ∈ [0, n.º de mutaciones – 1] HACER:
solución a perturbar = mutarSolucion(solución a perturbar)
FIN PARA CADA
DEVOLVER solución a perturbar
}

Ésta perturbación hace uso del operador de obtención de vecinos implementado para el enfriamiento simulado (mutarSolucion) y lo aplicamos tantas veces como se especificó en el guión (0.1 * n.º de elementos a elegir en una solución*).*

D) Algoritmo Híbrido ILS-ES

El algoritmo ILS-ES tiene la misma composición que el ILS estándar, con la única diferencia que el algoritmo de búsqueda por trayectorias simples considerado para refinar las soluciones iniciales será el enfriamiento simulado en lugar de la búsqueda local empleada hasta ahora.

El pseudocódigo, al ser casi idéntico que el del algoritmo ILS salvo por el cambio de usar la función búsquedaLocal por enfriamientoSimulado, no vamos a incluirlo (además no se exige en el guión).

Destacamos que los resultados, usando los parámetros ajustados para el algoritmo de enfriamiento simulado de forma independiente, no eran suficientemente buenos, así hemos adaptado dichos parámetros como sigue:

- máx. vecinos = tamaño de solución del problema (reducimos 10 veces el tamaño anterior).
- máx. éxitos = 0.1 * máx. vecinos
- temperatura final = 10^{-3}
- temperatura inicial = diversidad de la solución inicial * 0.3 / (- log(0.3))
- n.º máx de evaluaciones = 10.000
- enfriamiento de proporcional con alpha = 0.95

Por otro lado hemos inicializado las iteraciones a 10 y el n.º máx. de evaluaciones a 10.000 como se sugería en el guión de la práctica.

E) Búsqueda Local (Práctica 1 adaptada)

Como indica el guión, incluiremos el pseudocódigo del algoritmo de búsqueda local (que usamos el implementado en la práctica 1):

<u>Parámetros:</u> matriz de distancias (m), solución (representación binaria), n.º máx. de evaluaciones.
<u>Resultado:</u> solución calculada y diversidad de la solución.
busquedaLocal*{*
solución = pasar a representación en enteros la solución valida
parada = false
evaluaciones = 0
MIENTRAS parada = false y evaluaciones < n.º máx. de evaluaciones HACER:
n.º evaluaciones restantes = n.º máx. de evaluaciones - evaluaciones
parada = exploracionVecindario(s, evaluaciones, n.º evaluaciones restantes, m)
FIN MIENTRAS
solución = pasar a representación binaria la solución valida
DEVOLVER solución y su diversidad
}

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 evaluaciones en la ejecución (en nuestro caso este número será 10.000 evaluaciones).

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:

<u>Parámetros:</u> solución, evaluaciones (se va actualizando), n.º evaluaciones restantes y
matriz_de_distancias (m).
<u>Resultado:</u> booleano que indica si se ha mejorado, solución mejorada y evaluaciones actualizadas.
exploracionVecindario*{*
solución = ordenaSolucionPorContribucion(·)
limite_evaluaciones = n.º evaluaciones restantes / n.º de elementos que forman la solución

```
        i = 0
        e = 0
        MIENTRAS i < n.º de elementos que forman la solución HACER:
                elemento_actual = solución.vector[i]
                antigua_contribución =  distanciaAUnConjunto(elemento_actual, solución)
                k = 0
                índice_aleatorio = generarNumeroAleatorio(·)
                MIENTRAS k < limite_evaluaciones HACER:
                        SI índice_aleatorio ∉ solución HACER:
                                nueva_contribución  =  distanciaAUnConjunto(índice_aleatorio,  /
                                        / solución) – matriz[índice_aleatorio][elemento_actual]
                                SI nueva_contribución > antigua_contribución HACER:
                                        solución.vector[i] =  índice_aleatorio
                                        solución.diversidad = solución.diversidad +
                                                + nueva_contribución - antigua_contribución;
                                        DEVOLVER seguir_mejorando
                                FIN SI
                                k = k + 1
                                e = e + 1
                                SI e % n.º de elementos que forman la solución = 0 HACER:
                                        e = 0
                                        evaluaciones = evaluaciones + 1
                                FIN SI
                        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 4 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), FUENTES (donde se encuentran los ficheros creados para la práctica: ES.cpp donde se incluye el algoritmo de enfriamiento simulado, BMB.cpp donde se incluye el algoritmo de búsqueda multiarranque básica, ILS.cpp donde se incluye el algoritmo de búsqueda local reiterada y el fichero ILS-ES.cpp donde se incluye el algoritmo de búsqueda local reiterada hibrido) 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 cuatro algoritmos con algunos de los problemas de ejemplo.
- make Resultados: realiza las 120 ejecuciones necesarias para obtener los datos

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

Todos los ficheros deben 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 cuatro algoritmos:

A) Enfriamiento Simulado

ES				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	18213,2	19587,12891	7,01	0,011101
GKD-c_12_n500_m50	17753,3	19360,23633	8,30	0,020106
GKD-c_13_n500_m50	17908	19366,69922	7,53	0,025871
GKD-c_14_n500_m50	17843,2	19458,56641	8,30	0,018293
GKD-c_15_n500_m50	17816,1	19422,15039	8,27	0,017542
GKD-c_16_n500_m50	18113,9	19680,20898	7,96	0,023853
GKD-c_17_n500_m50	17956,5	19331,38867	7,11	0,020434
GKD-c_18_n500_m50	17813,6	19461,39453	8,47	0,014767
GKD-c_19_n500_m50	17977,3	19477,32813	7,70	0,021288
GKD-c_20_n500_m50	17859	19604,84375	8,91	0,018217
MDG-b_1_n500_m50	753490	778030,625	3,15	0,013635
MDG-b_2_n500_m50	755596	779963,6875	3,12	0,009588
MDG-b_3_n500_m50	746678	776768,4375	3,87	0,005824
MDG-b_4_n500_m50	745712	775394,625	3,83	0,006521
MDG-b_5_n500_m50	740626	775611,0625	4,51	0,005948
MDG-b_6_n500_m50	749027	775153,6875	3,37	0,005698
MDG-b_7_n500_m50	767180	777232,875	1,29	0,01107
MDG-b_8_n500_m50	748484	779168,75	3,94	0,006716
MDG-b_9_n500_m50	749791	774802,1875	3,23	0,005128
MDG-b_10_n500_m50	749290	774961,3125	3,31	0,007227
MDG-a_31_n2000_m200	112644	114139	1,31	0,539468
MDG-a_32_n2000_m200	112309	114092	1,56	0,311093
MDG-a_33_n2000_m200	112631	114124	1,31	0,347056
MDG-a_34_n2000_m200	112822	114203	1,21	0,513395
MDG-a_35_n2000_m200	112691	114180	1,30	0,39375
MDG-a_36_n2000_m200	112444	114252	1,58	0,335689
MDG-a_37_n2000_m200	112379	114213	1,61	0,352085
MDG-a_38_n2000_m200	112655	114378	1,51	0,287799
MDG-a_39_n2000_m200	112603	114201	1,40	0,293427
MDG-a_40_n2000_m200	112778	114191	1,24	0,384657

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

Media Desv:	4,2407025341716
Media Tiempo:	0,134241533333333

B) Búsqueda Multiarranque Básica

BMB				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	18277,2	19587,12891	6,69	0,361648
GKD-c_12_n500_m50	17767,8	19360,23633	8,23	0,372433
GKD-c_13_n500_m50	17961,2	19366,69922	7,26	0,363981
GKD-c_14_n500_m50	17906	19458,56641	7,98	0,342849
GKD-c_15_n500_m50	17866,6	19422,15039	8,01	0,345383
GKD-c_16_n500_m50	18155,1	19680,20898	7,75	0,432605
GKD-c_17_n500_m50	17989,6	19331,38867	6,94	0,456048
GKD-c_18_n500_m50	17860,9	19461,39453	8,22	0,417601
GKD-c_19_n500_m50	17987,1	19477,32813	7,65	0,377228
GKD-c_20_n500_m50	17878,3	19604,84375	8,81	0,348509
MDG-b_1_n500_m50	769644	778030,625	1,08	0,307025
MDG-b_2_n500_m50	766384	779963,6875	1,74	0,311444
MDG-b_3_n500_m50	767931	776768,4375	1,14	0,334376
MDG-b_4_n500_m50	767571	775394,625	1,01	0,363196
MDG-b_5_n500_m50	766933	775611,0625	1,12	0,321965
MDG-b_6_n500_m50	772367	775153,6875	0,36	0,314709
MDG-b_7_n500_m50	770543	777232,875	0,86	0,329191
MDG-b_8_n500_m50	773594	779168,75	0,72	0,311026
MDG-b_9_n500_m50	766350	774802,1875	1,09	0,31523
MDG-b_10_n500_m50	767466	774961,3125	0,97	0,346614
MDG-a_31_n2000_m200	113103	114139	0,91	12,9032
MDG-a_32_n2000_m200	112965	114092	0,99	13,8968
MDG-a_33_n2000_m200	113116	114124	0,88	12,339
MDG-a_34_n2000_m200	113134	114203	0,94	12,4059
MDG-a_35_n2000_m200	113009	114180	1,03	12,14
MDG-a_36_n2000_m200	113163	114252	0,95	12,6464
MDG-a_37_n2000_m200	113218	114213	0,87	12,0933
MDG-a_38_n2000_m200	112914	114378	1,28	11,1163
MDG-a_39_n2000_m200	113022	114201	1,03	11,3394
MDG-a_40_n2000_m200	113111	114191	0,95	12,7268

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

Media Desv:	3,24772133011917
Media Tiempo:	4,35600536666667

C) **Búsqueda Local Reiterada (ILS)**

ILS				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	18275,2	19587,12891	6,70	0,156464
GKD-c_12_n500_m50	17786,5	19360,23633	8,13	0,249534
GKD-c_13_n500_m50	17961,2	19366,69922	7,26	0,13693
GKD-c_14_n500_m50	17905,5	19458,56641	7,98	0,164449
GKD-c_15_n500_m50	17869,3	19422,15039	8,00	0,181011
GKD-c_16_n500_m50	18136,3	19680,20898	7,84	0,155159
GKD-c_17_n500_m50	17990,6	19331,38867	6,94	0,185069
GKD-c_18_n500_m50	17871,6	19461,39453	8,17	0,202546
GKD-c_19_n500_m50	17989,4	19477,32813	7,64	0,146041
GKD-c_20_n500_m50	17898,6	19604,84375	8,70	0,191573
MDG-b_1_n500_m50	764599	778030,625	1,73	0,128036
MDG-b_2_n500_m50	767858	779963,6875	1,55	0,158811
MDG-b_3_n500_m50	772297	776768,4375	0,58	0,142409
MDG-b_4_n500_m50	768395	775394,625	0,90	0,151519
MDG-b_5_n500_m50	763651	775611,0625	1,54	0,206384
MDG-b_6_n500_m50	762426	775153,6875	1,64	0,131544
MDG-b_7_n500_m50	774977	777232,875	0,29	0,218603
MDG-b_8_n500_m50	773330	779168,75	0,75	0,149841
MDG-b_9_n500_m50	762162	774802,1875	1,63	0,168134
MDG-b_10_n500_m50	765320	774961,3125	1,24	0,154346
MDG-a_31_n2000_m200	113028	114139	0,97	6,05271
MDG-a_32_n2000_m200	113233	114092	0,75	5,71245
MDG-a_33_n2000_m200	113147	114124	0,86	6,28634
MDG-a_34_n2000_m200	113263	114203	0,82	6,15542
MDG-a_35_n2000_m200	113270	114180	0,80	5,91569
MDG-a_36_n2000_m200	113355	114252	0,79	6,2781
MDG-a_37_n2000_m200	113491	114213	0,63	6,90263
MDG-a_38_n2000_m200	113408	114378	0,85	5,51517
MDG-a_39_n2000_m200	112938	114201	1,11	6,25589
MDG-a_40_n2000_m200	113267	114191	0,81	6,34422

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

Media Desv:	3,25305230015187
Media Tiempo:	2,15990076666667

D) **Hibridación de ILS y ES (ILS-ES)**

ILS-ES				
Caso	Coste obtenido	Mejor coste	Desv.	Tiempo (s.)
GKD-c_11_n500_m50	17867,2	19587,12891	8,78	0,042126
GKD-c_12_n500_m50	17525,9	19360,23633	9,47	0,059089
GKD-c_13_n500_m50	17720,1	19366,69922	8,50	0,039997
GKD-c_14_n500_m50	17606,4	19458,56641	9,52	0,050956
GKD-c_15_n500_m50	17546,4	19422,15039	9,66	0,040478
GKD-c_16_n500_m50	17954,3	19680,20898	8,77	0,037426
GKD-c_17_n500_m50	17820,5	19331,38867	7,82	0,041972
GKD-c_18_n500_m50	17641,2	19461,39453	9,35	0,064083
GKD-c_19_n500_m50	17710,5	19477,32813	9,07	0,040103
GKD-c_20_n500_m50	17628,6	19604,84375	10,08	0,042594
MDG-b_1_n500_m50	728080	778030,625	6,42	0,040551
MDG-b_2_n500_m50	743507	779963,6875	4,67	0,049565
MDG-b_3_n500_m50	740066	776768,4375	4,73	0,037143
MDG-b_4_n500_m50	738161	775394,625	4,80	0,076769
MDG-b_5_n500_m50	733844	775611,0625	5,39	0,045109
MDG-b_6_n500_m50	743850	775153,6875	4,04	0,041605
MDG-b_7_n500_m50	737802	777232,875	5,07	0,036554
MDG-b_8_n500_m50	731734	779168,75	6,09	0,061203
MDG-b_9_n500_m50	742774	774802,1875	4,13	0,038062
MDG-b_10_n500_m50	734460	774961,3125	5,23	0,038845
MDG-a_31_n2000_m200	110778	114139	2,94	0,924305
MDG-a_32_n2000_m200	110504	114092	3,14	0,833169
MDG-a_33_n2000_m200	110820	114124	2,90	0,833646
MDG-a_34_n2000_m200	110462	114203	3,28	0,835134
MDG-a_35_n2000_m200	110457	114180	3,26	0,818761
MDG-a_36_n2000_m200	110549	114252	3,24	0,891889
MDG-a_37_n2000_m200	110645	114213	3,12	0,880418
MDG-a_38_n2000_m200	110302	114378	3,56	0,904199
MDG-a_39_n2000_m200	110337	114201	3,38	0,8977
MDG-a_40_n2000_m200	110523	114191	3,21	0,874805

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

Media Desv:	5,78783877637204
Media Tiempo:	0,320608533333333

Conclusiones:

Lo primero que podemos observar, dados los resultados anteriores es que los 4 métodos llegan a soluciones bastante notables. Más concretamente para los algoritmos ILS y BMB los resultados son un poco mejores que el resto (plasmaremos una comparativa entre ambos métodos a continuación).

Si juntamos y resumimos la información anterior sobre desviaciones y tiempos de todos los algoritmos, tenemos:

	Algoritmo					
	ES	BMB	ILS	ILS-ES	BL	Greddy
Desv.	4,2407	3,247721	3,2531	5,7878	3,4783	4,44083647
Tiempo (s.)	0,1342	4,356005	2,1599	0,3206	11,3907	0,1761682

Hemos incluído la Búsqueda Local y el algoritmo Greddy de la Práctica 1, tal y como pedía el guión.

Como venimos analizando durante las prácticas anteriores, la búsqueda local ha resultado bastante efectiva con las instancias de nuestro problema dado. Sin embargo suele caer en óptimos locales y es lógico que no se alcance el óptimo del problema. Es decir, analiza en profundidad una solución (buena explotación), pero tiene muy poca diversidad (baja exploración).

Para proporcionar mayor diversidad o exploración hemos implementado el algoritmo del enfriamiento simulado, que permite movimientos de empeoramiento de la solución actual. Los resultados no mejoran a los obtenidos por la búsqueda local. En la comparación con greedy podemos decir que ES ofrece mejoras debido al hecho de no realizar una búsqueda de tipo voraz, sino centrada en el balance entre exploración y explotación del espacio de soluciones enfriando temperatura.

Otra forma de proporcionar mayor exploración de forma sencilla a la búsqueda local es partir de varias puntos iniciales (en nuestro caso aleatorios) y realizar una búsqueda local para cada uno de ellos. Es lo que hace el algoritmo de BMB. Sin embargo al estar limitados por el número de evaluaciones de la función objetivo en 100.000, hemos de reducir el número de evaluaciones para cada una de las búsquedas locales efectuadas, lo que provoca que perdamos explotación. Así al reajustar estos dos parámetros (explotación y exploración) en la búsqueda local con sistema multiarranque, hemos obtenido unos resultados un poco mejores que la búsqueda local original. Ésto nos hace pensar que con poco que aumente el límite de evaluaciones, el resultado será mejor, pues con solo 10.000 evaluaciones de una búsqueda local para un punto concreto de los elegidos hemos mejorado las 100.000 evaluaciones de la búsqueda local para el punto original. Así si hubiésemos permitido que siguiese explotando ese punto, el resultado hubiese sido mejor.

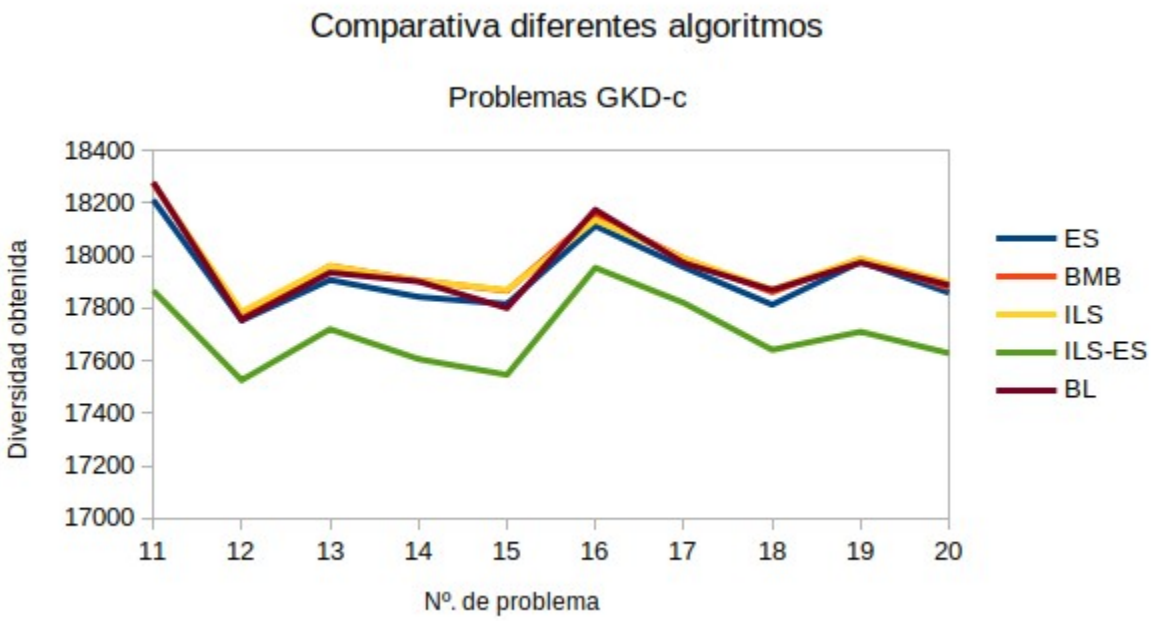
La tercera alternativa planteada (Búsqueda Local Reiterada) ha sido realizar una búsqueda local de un elemento aleatorio y cuando se haya realizado la explotación, modificarlo de forma brusca para permitir una exploración del problema y volver a realizar explotaciones de los puntos resultantes en cada iteración. Así de los algoritmos de la Práctica 3 éste es sin duda, junto a la BMB, el que proporciona los resultados más satisfactorios. Era esperado porque la Búsqueda Local en sí tenía buenos valores de la función objetivo en los tres ejemplos y la ésta nueva estrategia de mutación le iba a permitir aún más aumentar su explotación.

El último método planteado consistía en hibridar el enfriamiento simulado con el algoritmo ILS, es decir, aplicar un Enfriamiento Simulado Reiterado. Era esperable que los resultados fuesen mucho peores, ya que el Enfriamiento Simulado resultaba mucho menos eficaz que la Búsqueda Local para las instancias de nuestro problema.

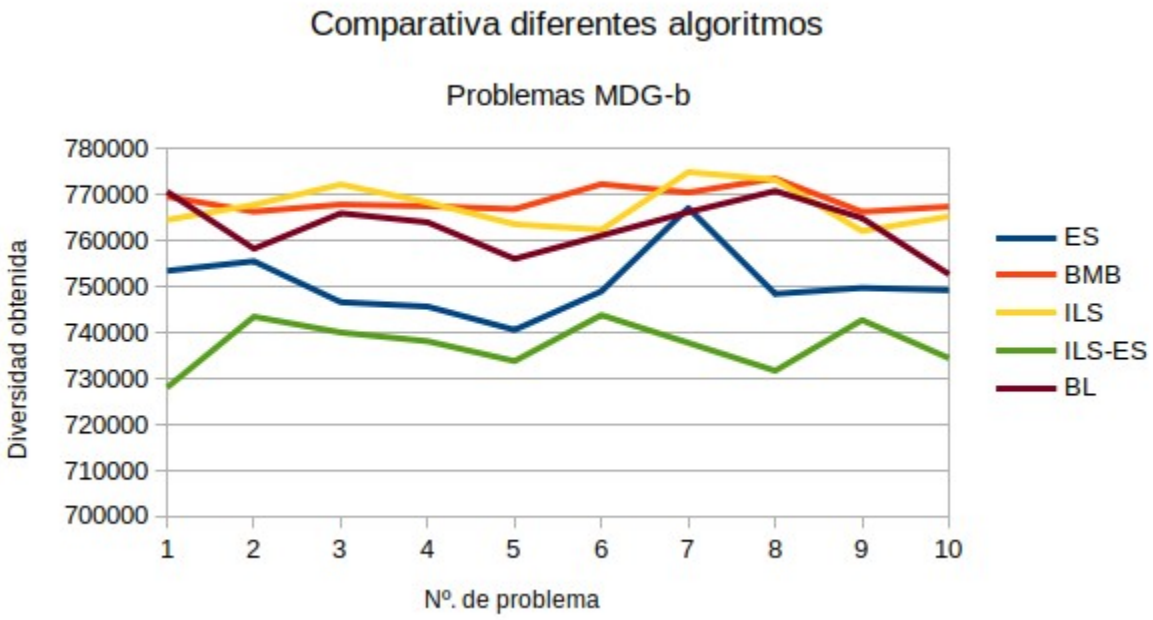
Desgranando los resultados, podemos analizar uno a uno los resultados obtenidos para cada problema con los 4 nuevos algoritmos junto con el algoritmo de la búsqueda local que es la base de nuestro intento de mejora. Es la tabla que justamente tenemos debajo. Vemos que salvo casos concretos, en general los algoritmos de ILS y BMB mejoran los resultados obtenidos por el algoritmo búsqueda local, mientras que los demás son peores que éstos.

	Algoritmo				
	ES	BMB	ILS	ILS-ES	BL
Caso	Diversidad				
GKD-c_11_n500_m50	18213,2	18277,2	18275,2	17867,2	18280
GKD-c_12_n500_m50	17753,3	17767,8	17786,5	17525,9	17755,6
GKD-c_13_n500_m50	17908	17961,2	17961,2	17720,1	17935,8
GKD-c_14_n500_m50	17843,2	17906	17905,5	17606,4	17902,2
GKD-c_15_n500_m50	17816,1	17866,6	17869,3	17546,4	17800,6
GKD-c_16_n500_m50	18113,9	18155,1	18136,3	17954,3	18175,5
GKD-c_17_n500_m50	17956,5	17989,6	17990,6	17820,5	17972,5
GKD-c_18_n500_m50	17813,6	17860,9	17871,6	17641,2	17869,2
GKD-c_19_n500_m50	17977,3	17987,1	17989,4	17710,5	17974
GKD-c_20_n500_m50	17859	17878,3	17898,6	17628,6	17887,9
MDG-b_1_n500_m50	753490	769644	764599	728080	770730
MDG-b_2_n500_m50	755596	766384	767858	743507	758308
MDG-b_3_n500_m50	746678	767931	772297	740066	766037
MDG-b_4_n500_m50	745712	767571	768395	738161	764054
MDG-b_5_n500_m50	740626	766933	763651	733844	756103
MDG-b_6_n500_m50	749027	772367	762426	743850	761191
MDG-b_7_n500_m50	767180	770543	774977	737802	766263
MDG-b_8_n500_m50	748484	773594	773330	731734	770877
MDG-b_9_n500_m50	749791	766350	762162	742774	764975
MDG-b_10_n500_m50	749290	767466	765320	734460	752744
MDG-a_31_n2000_m200	112644	113103	113028	110778	113640
MDG-a_32_n2000_m200	112309	112965	113233	110504	113234
MDG-a_33_n2000_m200	112631	113116	113147	110820	113092
MDG-a_34_n2000_m200	112822	113134	113263	110462	113422
MDG-a_35_n2000_m200	112691	113009	113270	110457	112753
MDG-a_36_n2000_m200	112444	113163	113355	110549	113310
MDG-a_37_n2000_m200	112379	113218	113491	110645	113501
MDG-a_38_n2000_m200	112655	112914	113408	110302	113302
MDG-a_39_n2000_m200	112603	113022	112938	110337	112674
MDG-a_40_n2000_m200	112778	113111	113267	110523	112959

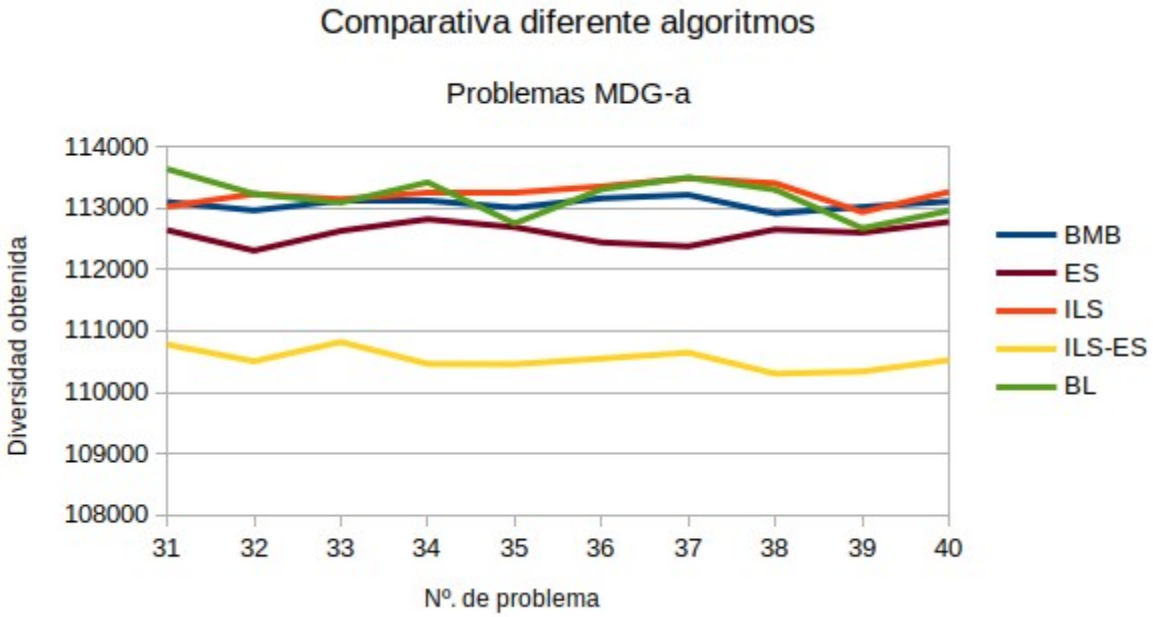
Para visualizarlos de forma correcta, hemos incluido las siguientes tablas con los resultados de cada problema agrupados por cada clase de problemas:



Vemos que salvo el ILS-ES, los demás obtienen resultados bastante análogos.



Realmente es para éstas instancias el ejemplo claro donde BMB e ILS les ganan terreno a los otros 3 algoritmos.



Podemos asegurar así para nuestro problema, que los algoritmos basados en trayectorias simples (búsqueda local y enfriamiento simulado) son un poco peores que los algoritmos basados en trayectorias múltiples aunque no podemos decir que en general ésto ocurre para cualquier algoritmo de los dos grupos, puesto que el ILS-ES empeora cualquier algoritmo de los otros 4 implementados y está basado en trayectorias múltiples. Tenemos así un claro ejemplo contradictorio.

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/

- PRADO y página web de la asignatura.