

PRÁCTICA 1

METAHEURÍSTICAS

PROBLEMA 1 : MDD



UNIVERSIDAD
DE GRANADA

David Muñoz Sánchez 07256819C
dmunozs14@correo.ugr.es Grupo A3

Índice

Índice	2
Breve descripción del problema	2
Breve descripción de los algoritmos empleados	2
Pseudocódigo	3
Algoritmo de comparación (Greedy)	3
Operador de generación de vecino	3
Factorización de la BL	4
Generación de soluciones aleatorias (BL)	4
Procedimiento para el desarrollo de la práctica	4
Algoritmo Greedy	5
BL	5
Experimentos y análisis de resultados	5
Greedy	5
BL	8

Breve descripción del problema

Tenemos un conjunto de distancias entre n puntos y debemos obtener el subconjunto con tamaño m cuya dispersión sea mínima.

Para ello, se usan dos métodos, el algoritmo Greedy y la BL. La solución no puede contener elementos repetidos y el orden de los elementos no es importante.

Breve descripción de los algoritmos empleados

Para este problema, se usa un algoritmo Greedy (voraz) y una búsqueda basada en trayectorias. En el primer caso, se trata de dado un nodo inicial aleatorio, encontremos la solución de tamaño m dentro de nuestros elementos (n elementos), que minimice la dispersión entre los puntos. La elección del segundo elemento también ha sido aleatoria puesto que cualquier conjunto de dos elementos tiene dispersión igual a 2, sea la distancia que sea entre ambos.

En el algoritmo Greedy, por cada elemento que tenemos que introducir en la solución, recorremos todos los elementos

seleccionables y vemos cual empeora menos o cual mejora la dispersión anterior, para finalmente añadirlo a nuestra solución.

En cuanto a BL, partimos de una solución aleatoria y la vamos mejorando generando vecinos, es decir, generando una nueva solución intercambiando un elemento, y sustituimos si mejora la dispersión de la nuestra.

Pseudocódigo

Algoritmo de comparación (Greedy)

```
Si seleccionables[i] != -1 entonces
    elegidos[] ← seleccionables[i]
    new_disp ← nueva_dispersion
    resto ← new_disp - min_disp

    Si primera_iteracion entonces
        resto_anterior ← resto
        elegido ← seleccionables[i]
        indice_borrar ← i
        first_ite ← false

    Si resto menor que resto_anterior o new_disp es menor que
    min_disp entonces
        resto_anterior ← resto
        elegido ← seleccionables[i]
        indice_borrar ← i

    elegidos.popback
```

Operador de generación de vecino

```
resultado[] ← solucion_actual
primer_sel ← Random::get(0,1000);
//Aquí vendría la aceptación
Mientras ((resultado[iteracion] == (resultado[iteracion] +
primer_sel) % total_elementos) or seleccionables[primer_sel] == -1)
    entonces
        primer_sel ← Random::get(0,1000);
        seleccionables[solucion_actual[iteracion]] ←
        solucion_actual[iteracion]
        resultado[iteracion] ← ((resultado[iteracion] + primer_sel) %
        total_elementos)
        seleccionables[resultado[iteracion]] ← -1
```

Factorización de la BL

La factorización se encuentra en el método `fitness_factorizado`.

```
solucion ← solucion_actual

for i ← 0 si i < m, i++ entonces
    Si i != indice_cambio entonces
        distancias_acumuladas.at(i) -=
matriz_distancias[solucion_actual[indice_cambio]][solucion_actual[i]
]

for i ← 0 si i < m, i++ entonces
    Si i != indice_cambio entonces
        distancias_acumuladas.at(i) +=
matriz_distancias[solucion_actual[indice_cambio]][solucion_actual[i]
]

    suma += distancias_acumuladas.at(i)
```

Generación de soluciones aleatorias (BL)

```
primer_sel ← Random::get(0,n-1)
solucion_inicial[] ← primer_sel
segundo_sel ← Random::get(0,n-1)
seleccionado ← segundo_sel
seleccionado_anterior ← primer_sel

Mientras seleccionado == seleccionado_anterior or
seleccionables.at(seleccionado) == -1 entonces
    seleccionado ← Random::get(0,n-1)
    Si seleccionado != seleccionado_anterior and
seleccionables.at(seleccionado) != -1 entonces
        solucion_inicial[] ← seleccionado
        solucion_actual[] ← seleccionado
        mejor_solucion[] ← seleccionado
```

Procedimiento para el desarrollo de la práctica

Lo primero, que es exactamente igual para los dos algoritmos, es leer el fichero con los datos de las distancias entre elementos. Hay que tener en cuenta que en el fichero se nos proporciona una matriz triangular, pero como la distancia euclídea de 0 a 1 es la misma que de 1 a 0, podemos completar la matriz entera para que después sea más fácil el acceso. La lectura la he implementado siguiendo implementaciones de sobrecarga del operador `>>` en la asignatura MP.

Algoritmo Greedy

Una vez seleccionados los dos primeros elementos que conformarán la solución, por cada elemento que falta, debemos recorrer todos los nodos seleccionables y hacemos la comparación arriba detallada en pseudocódigo. En cada iteración del while (el encargado de hacer m iteraciones), min_disp se actualiza a la dispersión de la solución que llevamos hasta el momento.

BL

Partimos de una solución aleatoria, vamos generando vecinos y comprobando si ese vecino mejora nuestra solución. La dispersión hay que calcularla de forma factorizada.

Experimentos y análisis de resultados

Para ambos algoritmos, la semilla para inicializar los números aleatorios varía entre 1 y 5 (ambos incluidos). La semilla se introduce como un argumento al ejecutar el programa (al igual que los ficheros con la matriz distancia).

Greedy

Algoritmo Greedy			
Caso	Coste medio obtenido	Desv	Tiempo (ms)
GKD-b_1_n25_m2	0,0000	0,00	No medido
GKD-b_2_n25_m2	0,0000	0,00	No medido
GKD-b_3_n25_m2	0,0000	0,00	No medido
GKD-b_4_n25_m2	0,0000	0,00	No medido
GKD-b_5_n25_m2	0,0000	0,00	No medido
GKD-b_6_n25_m7	45,2000	71,86	9,73E-02
GKD-b_7_n25_m7	44,6364	68,41	1,33E-01
GKD-b_8_n25_m7	49,8935	66,41	1,12E-01

GKD-b_9_n25_m7	55,4732	69,23	9,80E-02
GKD-b_10_n25_m7	52,2787	55,50	1,86E-01
GKD-b_11_n50_m5	16,2448	88,14	6,94E-02
GKD-b_12_n50_m5	19,3569	89,04	1,28E-01
GKD-b_13_n50_m5	12,3634	80,89	8,29E-02
GKD-b_14_n50_m5	15,9320	89,56	6,63E-02
GKD-b_15_n50_m5	24,7167	88,46	7,43E-02
GKD-b_16_n50_m15	208,0848	79,46	3,15E+00
GKD-b_17_n50_m15	163,5274	70,58	1,19E+00
GKD-b_18_n50_m15	109,3472	60,50	1,33E+00
GKD-b_19_n50_m15	153,1096	69,69	1,07E+00
GKD-b_20_n50_m15	130,7246	63,50	2,58E+00
GKD-b_21_n100_m10	71,8374	80,75	1,63E+00
GKD-b_22_n100_m10	68,1300	79,94	8,48E-01
GKD-b_23_n100_m10	77,0351	80,08	8,70E-01
GKD-b_24_n100_m10	53,7360	83,92	7,85E-01
GKD-b_25_n100_m10	63,5062	72,92	1,37E+00
GKD-b_26_n100_m30	465,6178	63,76	1,51E+01
GKD-b_27_n100_m30	419,9578	69,74	1,32E+01
GKD-b_28_n100_m30	423,2366	74,87	1,79E+01
GKD-b_29_n100_m30	302,7410	54,60	1,23E+01
GKD-b_30_n100_m30	394,9364	67,72	1,34E+01
GKD-b_31_n125_m12	100,0262	88,26	1,75E+00
GKD-b_32_n125_m12	54,4834	65,51	1,65E+00
GKD-b_33_n125_m12	110,2757	83,20	1,70E+00

GKD-b_34_n125_m12	68,6523	71,61	3,22E+00
GKD-b_35_n125_m12	77,6521	76,67	2,44E+00
GKD-b_36_n125_m37	463,7436	66,48	3,19E+01
GKD-b_37_n125_m37	468,0462	57,51	2,93E+01
GKD-b_38_n125_m37	494,4734	61,99	3,11E+01
GKD-b_39_n125_m37	473,3156	64,38	3,13E+01
GKD-b_40_n125_m37	383,5796	53,54	2,91E+01
GKD-b_41_n150_m15	83,5286	72,05	6,73E+00
GKD-b_42_n150_m15	118,5252	77,40	6,90E+00
GKD-b_43_n150_m15	139,0804	80,76	5,33E+00
GKD-b_44_n150_m15	122,8222	78,88	4,46E+00
GKD-b_45_n150_m15	103,7899	73,24	3,66E+00
GKD-b_46_n150_m45	584,6604	61,05	5,89E+01
GKD-b_47_n150_m45	492,0892	53,54	5,84E+01
GKD-b_48_n150_m45	437,4452	48,17	6,09E+01
GKD-b_49_n150_m45	626,1346	63,84	5,85E+01
GKD-b_50_n150_m45	649,5108	61,69	5,99E+01

Media Desv: **63,99**

Media Tiempo: **1,15E+01** (ms)

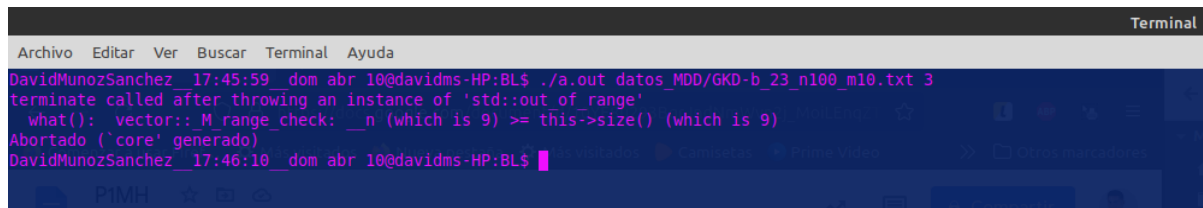
En los cinco primeros casos de prueba no se ha medido el tiempo ya que el tiempo en mi programa mide lo que tarda el algoritmo en rellenar el vector solución. Como cuando hay dos elementos sean cuáles sean la dispersión es 0, no hace falta entrar en el algoritmo.

El algoritmo presenta bastante desviación con respecto a la mejor solución conocida, lo cual es normal ya que estamos partiendo de dos elementos totalmente aleatorios.

De cada caso de prueba se han hecho 5 ejecuciones (con semilla del 1 al 5), y en la tabla se refleja la media de lo obtenido.

BL

La búsqueda local no me funciona correctamente, me surge el siguiente fallo:

A screenshot of a terminal window with a dark background. The title bar at the top says "Terminal". The terminal shows a command prompt where a user named DavidMunozSanchez runs a program. The program outputs a message about a terminate call and then crashes with a core dump. The error message is: "terminate called after throwing an instance of 'std::out_of_range'", followed by "what(): vector::M_range_check: __n (which is 9) >= this->size() (which is 9)", and "Abortado ('core' generado)". The prompt then shows the user's name and the time: "DavidMunozSanchez_17:46:10_dom abr 10@davidms-HP:BL\$".

```
DavidMunozSanchez_17:45:59_dom abr 10@davidms-HP:BL$ ./a.out datos_MDD/GKD-b_23_n100_m10.txt 3
terminate called after throwing an instance of 'std::out_of_range'
what(): vector::M_range_check: __n (which is 9) >= this->size() (which is 9)
Abortado ('core' generado)
DavidMunozSanchez_17:46:10_dom abr 10@davidms-HP:BL$
```

Además, previo a este fallo, he tenido bastante problemas con los vectores porque al igualarlos o al construir otro a partir de uno que ya tengo, se me cambiaban los tamaños. Es por eso que en el código hay bastantes resizes, que han sido usados para ver si se arreglaba el error pero persiste.

No obstante, hago la entrega porque pienso que lo que he hecho para la búsqueda local tiene bastante sentido.

Si bien no se realizan las iteraciones pedidas (en mi archivo) ya que leí después que eran 100000, lo que interesa es la forma de actualizar las soluciones, que es idéntica a la que se expone en los apuntes de clase. La aceptación del vecino se hace en la función `genera_vecino`, que nunca generará un vecino inválido. Además, el método `fitness_factorizado` actualiza el vector `distancias_acumuladas` mediante factorización y después en base a ese vector calcula la nueva dispersión.

Por tanto, todo me hace pensar que estoy haciendo algo con un vector que no debo, pero, dadas las circunstancias, tengo que entregarlo así.