Metaheurísticas (2018-2019)
Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Práctica 1: Algoritmo Greedy y de Búsqueda Local para el Problema de Máxima Diversidad (MDP)

Víctor Bricio Blázquez 47424585B
vbricio@correo.ugr.es
Grupo 2: Miércoles de 17:30h a 19:30h

1 de abril de 2019

Índice

1. Descripción del problema

El problema se trata de elegir los m elementos de entre una familia de n elementos que cumplan que la distancia entre estos m elementos es máxima.
Una de las partes más importantes de este enunciado es el concepto de distancia, pues se pueden tomar muchos tipos de distancia para atacar este problema. En este caso se nos pide que la distancia a tomar sea la suma de todas las distancias entre todos los elementos de nuestra solución (denominada MaxSum).
Llega ahora el concepto de solución, diremos que un conjunto de números es solución si cumple las siguientes características:

- Contiene exactamente m elementos.
- No existen elementos repetidos.
- El orden de los elementos no es relevante.

2. Descripción de las estructuras de datos

Para afrontar este problema he utilizado un lector de datos que convierte los archivos que se nos proporcionan en una matriz triangular, dejando la diagonal y el otro triángulo de la matriz como ceros.
He desarrollado una serie de vectores que son de notable interés:

- Solución: En este vector se estará construyendo una solución o será en sí una solución.
- Usados: En este vector estarán todos los elementos que se hayan probado para mejorar una solución y no la hayan mejorado.
- Mundo: En este vector estarán todos los elementos que no estén en la solución o en usados.

La función objetivo (denominada score) suma las distancias que relacionan todos los elementos de la solución con todos los elementos de la solución. Tal y cómo lo tengo diseñado estaría sumando todo dos veces y estaría sumando la distancia entre un elemento y él mismo. Pero el hecho de que la diagonal y el otro triángulo de la matriz estén llenas de ceros soluciona estos problemas.
Existe una función, denominada buscaPos, a la que se le pasa un parámetro. Esta función busca la posición del elemento que tiene como parámetro en el vector mundo, generalmente, para eliminarlo de ahí.

3. Búsqueda Local

El algoritmo de Búsqueda Local toma una solución inicial y va elaborando vecinos para compararlos con la solución actual y ver cual de las dos soluciones es mejor. Va haciendo esto hasta que realiza 50000 iteraciones o no hay un vecino mejor.
Aquí ha aparecido un nuevo concepto, el de vecino. En este caso un vecino será un conjunto de números idéntico a la solución actual con el siguiente cambio: uno de los elementos de la solución actual será modificado por uno de los elementos que no están en la solución actual.
Para efectuar el algoritmo de Búsqueda Local he seguido los siguientes pasos:

- Calculo el elemento que añade más a la solución actual y lo intercambio de manera aleatoria con uno de los elementos de la solución actual.
    - En caso de que este intercambio no sea beneficioso para la solución actual: Elimino este elemento que pretendía mejorar la solución de **mundo** y lo introduzco en **usados**.
    - En caso de que este intercambio sea beneficioso para la solución actual: Elimino este elemento, que ha mejorado la solución, de **mundo** y limpio **usados** (la operación de intercambio ya elimina de la solución el elemento desechado).
- Esto se repite hasta que se realicen 50000 iteraciones o hasta que **mundo** esté vacío.

Para elegir el elemento aleatorio de la solución para intercambiar, simplemente utilizo la función **shuffle** de la biblioteca **algorithm** y tomo el elemento 0.

Ahora voy a complementar la información que he dado sobre la función de intercambio para que se entienda mejor lo que hace. A parte de hacer lo que ya he comentado lo que realiza es una comparación de las puntuaciones de la solución actual y de la solución actual con el intercambio (por medio de una función denominada **remakeScore**).
En caso de que el intercambio no sea beneficioso no hace nada, en cambio, si el intercambio sí es beneficioso para la solución esta misma función se encarga de eliminar el elemento desechado y de introducir el nuevo elemento. Además de cambiar la variable global que controla la puntuación total de la solución actual.
Esta comparación se hace utilizando el método de Factorización del Movimiento de Intercambio, que consiste en no calcular toda la puntuación de la nueva solución, sino en conocer que aporta nuevo la nueva solución y que pierde la solución antigua, y con esto discriminar.

He elaborado también una función llamada distanciaAcumulada, a la que se le pasa un parámetro, esta obtiene la suma de todas las distancias entre el elemento del parámetro y el resto de la matriz, para así obtener el elemento más distante a todos los elementos de la matriz.

Por último, debo añadir que la solución inicial que utiliza el algoritmo de Búsqueda Local es aquella que proporciona el algoritmo greedy que tenga el caso que corresponde.

4. Algoritmo Greedy

El algoritmo greedy se basa en elegir en cada paso el mejor elemento, siendo en este caso el mejor aquel que cumple que esta más alejado de los que no están en la solución, aquellos que están en **mundo**, pues en este algoritmo no se usará usados.
Se sigue esta forma de actuar hasta que se llega a tener m elemento en la solución.
Este es un algoritmo sencillo que no tiene mucha explicación, así que esencia eso es lo que he programado:

- Elegir el elemento más distante a la solución actual (en este caso estoy sobrecargando la palabra solución, pues solo podrá considerarse una verdadera solución cuando tenga exactamente m elementos).
- Seguir este proceso hasta que, efectivamente, la solución, tenga exactamente m elementos.

5. Procedimiento para desarrollar la práctica

Para desarrollar la práctica he utilizado código implementado por mí desde el principio, pues todas las estructuras de datos y procedimientos a elaborar eran suficientemente sencillos como para hacerlos yo.
Para ejecutar la práctica basta con tener en una carpeta llamada **Tablas** las 30 tablas y compilarlo como un programa normal, se debe poner el siguiente comando:
g++ -o ../BIN/nombre_del_ejeutable p1.cpp
(Tengo entendido que se puede optimizar con la opción **-o2** o **-o3**, pero no consigo hacer que funcione).

6. Tablas

Algoritmo Greedy			
Caso	Coste obtenido	Desviación	Tiempo (s)
GKD-c_1_n500_m50	18990.2	2.5401	0.002641
GKD-c_2_n500_m50	19337.4	1.84813	0.002273
GKD-c_3_n500_m50	19242.5	1.55894	0.002299
GKD-c_4_n500_m50	19468.5	0.652951	0.002351
GKD-c_5_n500_m50	19076.4	2.68431	0.002388
GKD-c_6_n500_m50	19225.6	1.00918	0.002413
GKD-c_7_n500_m50	19059	2.43311	0.002445
GKD-c_8_n500_m50	19247.1	1.2326	0.002337
GKD-c_9_n500_m50	18967.9	1.32021	0.002421
GKD-c_10_n500_m50	19496.8	1.04824	0.002371
SOM-b_11_n300_m90	18229	12.1198	0.001279
SOM-b_12_n300_m120	32713	8.82919	0.001527
SOM-b_13_n400_m40	3492	25.0322	0.00147
SOM-b_14_n400_m80	14654	13.5763	0.001955
SOM-b_15_n400_m120	32232	11.2482	0.002313
SOM-b_16_n400_m160	57884	7.36633	0.002773
SOM-b_17_n500_m50	5517	22.7419	0.002424
SOM-b_18_n500_m100	22456	14.4794	0.003147
SOM-b_19_n500_m150	51188	9.51708	0.003894
SOM-b_20_n500_m200	90277	7.25982	0.004476
MDG-a_21_n2000_m200	99678	12.7614	0.069665
MDG-a_22_n2000_m200	99987	12.543	0.069771
MDG-a_23_n2000_m200	100043	12.3376	0.070309
MDG-a_24_n2000_m200	99544	12.7113	0.069193
MDG-a_25_n2000_m200	99534	12.7385	0.070314
MDG-a_26_n2000_m200	100567	11.9409	0.072164
MDG-a_27_n2000_m200	99984	12.554	0.070745
MDG-a_28_n2000_m200	99937	12.4573	0.070113
MDG-a_29_n2000_m200	100205	12.2025	0.07047
MDG-a_30_n2000_m200	100385	12.0949	0.077868

Búsqueda Local			
Caso	Coste obtenido	Desviación	Tiempo
GKD-c_1_n500_m50	19384	0.519328	0.151051
GKD-c_2_n500_m50	19555.3	0.742046	0.147882
GKD-c_3_n500_m50	19465.6	0.417345	0.148486
GKD-c_4_n500_m50	19541.6	0.280246	0.148788
GKD-c_5_n500_m50	19475.9	0.646656	0.147228
GKD-c_6_n500_m50	19361.4	0.30946	0.15029
GKD-c_7_n500_m50	19419.1	0.589824	0.146784
GKD-c_8_n500_m50	19417.5	0.358237	0.157478
GKD-c_9_n500_m50	19141.6	0.416592	0.146385
GKD-c_10_n500_m50	19640.2	0.320586	0.147096
SOM-b_11_n300_m90	20031	3.43248	0.089583
SOM-b_12_n300_m120	34848	2.87896	0.109101
SOM-b_13_n400_m40	4422	5.06655	0.079206
SOM-b_14_n400_m80	16088	5.11913	0.149735
SOM-b_15_n400_m120	35204	3.06468	0.220937
SOM-b_16_n400_m160	61219	2.02922	0.257481
SOM-b_17_n500_m50	6682	6.42767	0.154559
SOM-b_18_n500_m100	25082	4.47864	0.296601
SOM-b_19_n500_m150	54645	3.40628	0.441745
SOM-b_20_n500_m200	94799	2.61444	0.518693
MDG-a_21_n2000_m200	110264	3.49644	17.5395
MDG-a_22_n2000_m200	110523	3.3273	17.638
MDG-a_23_n2000_m200	110287	3.36129	17.3927
MDG-a_24_n2000_m200	110274	3.30235	17.2847
MDG-a_25_n2000_m200	110329	3.27448	18.5776
MDG-a_26_n2000_m200	110786	2.99289	18.2306
MDG-a_27_n2000_m200	110529	3.33135	18.3228
MDG-a_28_n2000_m200	110745	2.98972	18.4188
MDG-a_29_n2000_m200	110165	3.4758	18.7385
MDG-a_30_n2000_m200	110316	3.39851	17.8073

Algoritmo	Desviación	Tiempo
Greedy	9.09465	0.0237608
Búsqueda Local	2.53562	6.12532

7. Comparación y análisis de los resultados

Cómo podemos ver el algoritmo Greedy obtiene unos valores que podríamos considerar como decentes, pero aún y así tienen una varianza basta elevada. Sin embargo, el algoritmo de Búsqueda Local tiene una varianza mucho menor, con lo cual, esto implica que los valores son mucho más acercados al óptimo.
Aunque es fácil ver que los costes en tiempo del algoritmo Greedy son mucho menores que el tiempo del algoritmo de Búsqueda Local (soy consciente de que los tiempos que obtengo para la Búsqueda Local soy muy elevados y estoy trabajando en ello para arreglarlo).
Era esperable que el algoritmo de Búsqueda Local obtuviera mejores resultados que el algoritmo Greedy, pues busca más en profundidad soluciones potenciales, es decir, toma una solución que consideramos decente, y busca vecinos que la mejoren. Mientras que el Greedy solamente busca el mejor añadido en cada momento. Y de todos es sabido que considerar lo mejor en cada momento no es óptimo, pues por un camino inicialmente peor se puede encontrar soluciones que a la larga sean mucho más beneficiosas.

8. Referencias bibliográficas

Para realizar las prácticas me he basado en las explicaciones de las prácticas y la teoría de la asignatura y en c++ reference para abordar el funcionamiento de la clase vector.