Informe de práctica del Maximum Diversity Problem

ENRIQUE VIÑA ALONSO

Mayo 2022

1. Introducción

En el Maximum diversity problem se desea encontrar el subconjunto de elementos de diversidad máxima de un conjunto dado de elementos. Sea dado un conjunto $S = \{s_1, s_2, s_3, ..., s_n\}$ en el que cada elemento s_i es un vector de dimensión k. Sea asimismo d_{ij} la distancia entre los elementos i y j. Siendo $m < n$ el tamaño del subconjunto de solución del problema, se busca:

$$\text{Maximizar } z = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_i x_j$$

Sujeto a:

$$\sum_{i=1}^{n} x_i = m$$

$$x_i \in \{0, 1\} \qquad \forall i = 1, 2, ..., n$$

Donde:

$$x_i = \begin{cases} 1 & \text{si } s_i \text{ pertenece a la solución} \\ 0 & \text{en caso contrario} \end{cases}$$

La distancia d_{ij} depende de la aplicación real considerada, sin embargo para esta implementación se usará la distancia euclídea.

2. Descripción del código

En esta implementación se ha trabajado con el lenguaje de programación orientado a objetos $C\#$, sin hacer uso de ninguna librería externa.

2.1. Problem

Hemos creado una clase 'Problem', que será la encargada de extraer la información almacenada en los ficheros de datos. Se almacena el nombre del fichero, y cada uno de los vectores contenidos en el fichero, además se construye una matriz de distancias entre cada par de vectores.

2.2. Solution

En esta implementación se hace uso de una representación indexal de las soluciones, en concreto, tendremos un $HashSet < int > solution$ para los índices del conjunto de solución. La clase Solution también guarda los la lista de vectores totales del problema.

2.3. Algoritmos desarrollados

Se han implementado tres algoritmos y una búsqueda de entorno, que se detallarán a continuación:

2.3.1. Greedy

'*Greedy*' es una clase que implementa un algoritmo voraz con Restricted Candidate List. Para resolver el problema de forma puramente voraz, el tamaño de la RCL está asignado a 1 por defecto, de esta forma, el algoritmo se comporta como un algoritmo voraz tradicional.

2.3.2. Grasp

La clase '*Grasp*' implementa un algoritmo GRASP, el método 'Solve' toma como parámetro el tamaño de la RCL que tendrá la fase constructiva y el tamaño de la solución a encontrar. Las búsquedas locales implementadas son:

- Intercambio El movimiento realizado es intercambiar un elemento que esté presente en la solución por uno que no lo está.

2.3.3. Branch and Bound

La clase '*BranchAndBound*' implementa el algoritmo de Branch and Bound, durante el desarrollo del algoritmo se hace uso de una clase '*PartialSolution*' que almacena las soluciones parciales que se van generando a medida que se explora el árbol de soluciones.

3. Resultados

A continuación se presentan las mejores soluciones encontradas por los algoritmos implementados:

3.1. Algoritmo voraz

filename	n	dim	s_size	cost	milliseconds
max_div_15_2.txt	15	2	2	11.86	1
max_div_15_3.txt	15	3	2	13.27	0
max_div_20_2.txt	20	2	2	8.51	0
max_div_20_3.txt	20	3	2	11.80	0
max_div_30_2.txt	30	2	2	11.66	0
max_div_30_3.txt	30	3	2	13.07	0
max_div_15_2.txt	15	2	3	25.72	0
max_div_15_3.txt	15	3	3	30.32	0
max_div_20_2.txt	20	2	3	21.99	0
max_div_20_3.txt	20	3	3	30.87	0
max_div_30_2.txt	30	2	3	28.95	0
max_div_30_3.txt	30	3	3	33.83	0
max_div_15_2.txt	15	2	4	48.41	0
max_div_15_3.txt	15	3	4	59.76	0
max_div_20_2.txt	20	2	4	39.56	0
max_div_20_3.txt	20	3	4	56.52	0
max_div_30_2.txt	30	2	4	52.77	0
max_div_30_3.txt	30	3	4	63.51	0
max_div_15_2.txt	15	2	5	73.56	0
max_div_15_3.txt	15	3	5	94.75	0
max_div_20_2.txt	20	2	5	61.22	0
max_div_20_3.txt	20	3	5	92.81	0
max_div_30_2.txt	30	2	5	80.90	0
max_div_30_3.txt	30	3	5	99.50	0

Cuadro 1: Resultados del algoritmo Greedy.

3.2. Algoritmos GRASP

El algoritmo GRASP se ha calculado con varios parámetros diferentes, en concreto con 10 y 20 iteraciones máximas y con tamaño de la RCL 2 y 3. Sin embargo, pese a las variaciones de los parámetros los resultados son idénticos para cada uno de los problemas, en la siguiente tabla se muestran los resultados simplificados, eliminando repeticiones. Resultados de los algoritmos GRASP:

filename	n	dim	rcl	s_size	cost	milliseconds
max_div_15_2.txt	15	2	2	2	11.86	10
max_div_15_3.txt	15	3	2	2	13.27	7
max_div_20_2.txt	20	2	2	2	8.51	9
max_div_20_3.txt	20	3	2	2	11.80	9
max_div_30_2.txt	30	2	2	2	11.66	9
max_div_30_3.txt	30	3	2	2	13.07	11
max_div_15_2.txt	15	2	2	3	27.38	11
max_div_15_3.txt	15	3	2	3	31.87	11
max_div_20_2.txt	20	2	2	3	21.99	12
max_div_20_3.txt	20	3	2	3	30.87	13
max_div_30_2.txt	30	2	2	3	28.95	16
max_div_30_3.txt	30	3	2	3	34.28	19
max_div_15_2.txt	15	2	2	4	49.84	14
max_div_15_3.txt	15	3	2	4	59.76	12
max_div_20_2.txt	20	2	2	4	40.00	17
max_div_20_3.txt	20	3	2	4	56.68	16
max_div_30_2.txt	30	2	2	4	52.77	51
max_div_30_3.txt	30	3	2	4	63.69	27
max_div_15_2.txt	15	2	2	5	79.13	16
max_div_15_3.txt	15	3	2	5	96.10	13
max_div_20_2.txt	20	2	2	5	63.65	19
max_div_20_3.txt	20	3	2	5	92.81	19
max_div_30_2.txt	30	2	2	5	80.90	51
max_div_30_3.txt	30	3	2	5	99.58	79

Cuadro 2: Resultados del algoritmo GRASP con reinserción multirruta como búsqueda local.

3.3. Algoritmo Branch and Bound con DFS

Resultados del Algoritmo Branch and Bound:

filename	n	dim	s_size	cost	milliseconds	generated
max_div_15_2.txt	15	2	2	11.86	2	15
max_div_15_3.txt	15	3	2	13.27	1	15
max_div_20_2.txt	20	2	2	8.51	2	20
max_div_20_3.txt	20	3	2	11.80	2	20
max_div_30_2.txt	30	2	2	11.66	10	30
max_div_30_3.txt	30	3	2	13.07	10	30
max_div_15_2.txt	15	2	3	27.38	4	108
max_div_15_3.txt	15	3	3	31.87	4	109
max_div_20_2.txt	20	2	3	21.99	14	191
max_div_20_3.txt	20	3	3	30.87	15	191
max_div_30_2.txt	30	2	3	28.95	62	436
max_div_30_3.txt	30	3	3	34.28	34	437
max_div_15_2.txt	15	2	4	49.84	3	459
max_div_15_3.txt	15	3	4	59.76	4	456
max_div_20_2.txt	20	2	4	40.00	28	1109
max_div_20_3.txt	20	3	4	56.68	32	1123
max_div_30_2.txt	30	2	4	52.77	209	4005
max_div_30_3.txt	30	3	4	63.69	228	3936
max_div_15_2.txt	15	2	5	79.13	9	1341
max_div_15_3.txt	15	3	5	96.10	10	1341
max_div_20_2.txt	20	2	5	63.65	60	4235
max_div_20_3.txt	20	3	5	92.81	61	3698
max_div_30_2.txt	30	2	5	80.90	1245	24924
max_div_30_3.txt	30	3	5	99.58	1172	23232

Cuadro 3: Resultados del algoritmo Branch and Bound con búsqueda por profundidad.

3.4. Algoritmo Branch and Bound con SUF

Resultados del Algoritmo Branch and Bound:

filename	n	dim	s_size	cost	milliseconds	generated
max_div_15_2.txt	15	2	2	11.86	0	15
max_div_15_3.txt	15	3	2	13.27	0	15
max_div_20_2.txt	20	2	2	8.51	0	20
max_div_20_3.txt	20	3	2	11.80	0	20
max_div_30_2.txt	30	2	2	11.66	3	30
max_div_30_3.txt	30	3	2	13.07	3	30
max_div_15_2.txt	15	2	3	27.38	1	109
max_div_15_3.txt	15	3	3	31.87	1	108
max_div_20_2.txt	20	2	3	21.99	4	191
max_div_20_3.txt	20	3	3	30.87	4	191
max_div_30_2.txt	30	2	3	28.95	47	436
max_div_30_3.txt	30	3	3	34.28	58	437
max_div_15_2.txt	15	2	4	49.84	6	461
max_div_15_3.txt	15	3	4	59.76	6	456
max_div_20_2.txt	20	2	4	40.00	51	1110
max_div_20_3.txt	20	3	4	56.68	52	1123
max_div_30_2.txt	30	2	4	52.77	228	4005
max_div_30_3.txt	30	3	4	63.69	317	3936
max_div_15_2.txt	15	2	5	79.13	9	1342
max_div_15_3.txt	15	3	5	96.10	9	1343
max_div_20_2.txt	20	2	5	63.65	59	4243
max_div_20_3.txt	20	3	5	92.81	53	3698
max_div_30_2.txt	30	2	5	80.90	1309	24924
max_div_30_3.txt	30	3	5	99.58	1165	23232

Cuadro 4: Resultados del algoritmo Branch and Bound con búsqueda Smallest Upperbound First.

4. Conclusiones

Comparando los resultados obtenidos, podemos ver que, a persar de que el algoritmo Branch and Bound es un algoritmo exacto, no es capaz de encontrar soluciones de mayor calidad que el GRASP, además de ser más de 10 veces más lento.

Esto se debe al poco número de vectores del problema así como los pequeños tamaños de la solución, en este contexto el GRASP es capaz de encontrar la solución óptima, sin embargo, aumentando la cantidad de vectores a evaluar, deberíamos notar una diferencia entre la calidad de los valores de función objetivos de las soluciones encontradas por los algoritmos.