Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Informe de la Práctica 6 de DAA
Curso 2024-2025

Maximum diversity problem

Guillermo González Pineda

La Laguna, 6 de mayo de 2025

Índice general

Índice de Tablas

Capítulo 1

Introducción

1.1. Contexto

El *Maximum Diversity Problem* (MDP) es un problema clásico dentro del campo de la optimización combinatoria. Su objetivo es seleccionar un subconjunto de tamaño fijo a partir de un conjunto más amplio de elementos, de forma que la diversidad entre los seleccionados sea la máxima posible. Esta diversidad se mide habitualmente a través de la suma de las distancias entre todos los pares de elementos del subconjunto.

El MDP tiene múltiples aplicaciones en contextos donde es fundamental maximizar la heterogeneidad: desde la selección de ubicaciones geográficas dispersas, hasta la elección de elementos representativos en técnicas de muestreo, diseño de productos o planificación de rutas.

1.2. Motivación

La relevancia del MDP radica en su aplicabilidad a situaciones reales donde la diversidad resulta un criterio clave de calidad. A pesar de su formulación sencilla, se trata de un problema de naturaleza NP-dura, lo que implica que encontrar su solución óptima requiere un coste computacional elevado en instancias de tamaño medio o grande.

Por ello, resulta especialmente interesante explorar enfoques que, sin garantizar optimalidad, proporcionen soluciones de alta calidad en tiempos razonables. Esta práctica se enmarca en ese objetivo: estudiar y comparar distintas estrategias exactas y aproximadas para la resolución del MDP.

1.3. Resumen de la práctica

En el presente trabajo se ha abordado la resolución del MDP mediante la implementación de varios algoritmos:

- Un enfoque exacto basado en Ramificación y Poda (*Branch and Bound*), que explora el espacio de soluciones de forma estructurada y garantiza encontrar la solución óptima.
- Algoritmos aproximados como Greedy, GRASP (*Greedy Randomized Adaptive Search Procedure*) y búsqueda local por intercambio, que permiten encontrar buenas soluciones en tiempos muy inferiores.

La práctica se ha desarrollado en lenguaje C++, haciendo uso de principios de diseño orientado a objetos y estructuras modulares. Se han utilizado diversas instancias del problema, con distintos tamaños y dimensiones, para realizar una evaluación comparativa entre los algoritmos, considerando tanto la calidad de las soluciones como el coste computacional.

Capítulo 2

Descripción del Problema

2.1. Definición formal

El *Maximum Diversity Problem* (MDP) consiste en seleccionar un subconjunto de m elementos a partir de un conjunto original de n elementos, de manera que se maximice la diversidad entre los seleccionados. Esta diversidad se define como la suma de las distancias entre todos los pares de elementos dentro del subconjunto elegido.

Formalmente, sea $S = \{s_1, s_2, \ldots, s_n\}$ un conjunto de n elementos, donde cada elemento s_i se representa como un vector en un espacio de K dimensiones. El objetivo es encontrar un subconjunto $S' \subset S$ tal que $|S'| = m$ y:

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

$$\text{sujeto a} \quad \sum_{i=1}^{n} x_i = m, \quad x_i \in \{0,1\} \quad \forall i \in \{1, \ldots, n\}$$

donde $x_i = 1$ si el elemento s_i es seleccionado, y 0 en caso contrario. La distancia d_{ij} entre elementos se calcula mediante la distancia euclídea:

$$d_{ij} = \sqrt{\sum_{r=1}^{K} (s_{ir} - s_{jr})^2}$$

2.2. Parámetros del problema

Los principales parámetros que definen una instancia del MDP son:

- n: número total de elementos del conjunto original.
- m: número de elementos que se deben seleccionar ($m < n$).
- K: número de dimensiones del espacio en el que se representa cada elemento.
- D: matriz simétrica de distancias d_{ij} entre los elementos.

2.3. Función objetivo

La función objetivo busca maximizar la suma de distancias entre todos los pares de elementos seleccionados. Esta medida de diversidad refleja cuánto se "dispersan" los puntos dentro del subconjunto resultante, lo que permite capturar una solución representativa y heterogénea del espacio original.

2.4. Representación del problema (entrada/salida)

Las instancias del problema se representan mediante archivos de texto que siguen el siguiente formato:

- La primera línea contiene un número entero n, el número total de elementos.
- La segunda línea contiene un número entero K, el número de dimensiones.
- Las siguientes n líneas contienen las coordenadas de cada elemento, con K valores por línea separados por espacios o tabuladores.

Por ejemplo, una entrada con $n = 3$ y $K = 2$ podría tener el siguiente aspecto:

```
3
2
1.0 2.0
3.5 4.1
0.9 1.7
```

La salida esperada es el subconjunto seleccionado S', representado como un conjunto de índices, así como el valor total z alcanzado por la solución y, en caso de evaluación experimental, el tiempo de ejecución o el número de nodos explorados.

Capítulo 3

Estructura del Proyecto e Implementación

3.1. Organización del código

El proyecto se ha estructurado de forma modular, siguiendo buenas prácticas de diseño en `C++`, con separación entre declaraciones (`.h`) y definiciones (`.cpp`). La raíz del proyecto contiene subdirectorios con propósitos bien definidos:

- **`include/`**: contiene los ficheros de cabecera con la definición de las clases.
- **`src/`**: implementa la lógica de cada clase y los algoritmos.
- **`data/`**: incluye las instancias del problema en formato texto.
- **`build/`** y **`debug/`**: directorios generados por `CMake` para la compilación en distintas configuraciones.
- **`result/`**: carpeta destinada a almacenar salidas experimentales.
- **`main.cpp`**: fichero principal que permite ejecutar los algoritmos con distintas configuraciones.

La compilación se gestiona mediante `CMake`, lo que permite mantener la portabilidad y el control sobre el proceso de construcción.

3.2. Principales clases implementadas

A continuación se describen brevemente las clases más relevantes de la implementación:

ProblemInstance

Modela una instancia concreta del MDP. Almacena la matriz de distancias, el conjunto de puntos y los parámetros n y k. Incluye métodos para acceder a los datos y calcular distancias entre elementos.

Solution

Representa una solución candidata. Almacena los índices de los elementos seleccionados, el valor de la solución (suma de distancias), el tiempo de ejecución y otros parámetros útiles como m, tamaño de LRC o número de iteración. Incluye métodos para evaluación, formateo e impresión.

ParserData

Encargada de leer ficheros de entrada y construir objetos `ProblemInstance`. Realiza el procesamiento necesario para adaptar los datos a la estructura interna del programa.

GreedyAlgorithm, GraspAlgorithm, GraspRunner

Estas clases encapsulan los métodos heurísticos utilizados:

- `GreedyAlgorithm`: construcción determinista basada en el elemento más alejado del centroide y selección secuencial.
- `GraspAlgorithm`: versión aleatorizada de la heurística, utilizando listas LRC para introducir diversidad.
- `GraspRunner`: ejecuta múltiples veces el algoritmo GRASP y selecciona la mejor solución encontrada.

LocalSearch

Implementa la búsqueda local por intercambio 1-in 1-out. Explora el vecindario reemplazando elementos del conjunto actual para mejorar el valor de la solución.

BranchAndBoundSolver y BnBNode

Estas clases forman el núcleo del algoritmo exacto:

- `BranchAndBoundSolver`: controla la ejecución del algoritmo de ramificación y poda. Utiliza la heurística Greedy como cota inferior y expande nodos en orden descendente por cota superior.
- `BnBNode`: representa un nodo del árbol de búsqueda. Almacena los elementos seleccionados y restantes, así como las cotas calculadas (z1, z2, z3).

TableGenerator

Responsable de imprimir los resultados experimentales en un formato tabulado uniforme. Permite mostrar las soluciones obtenidas con distintos algoritmos y métricas (tiempo, calidad, nodos generados, etc.).

3.3. Ejecución

El comportamiento del programa se configura mediante el argumento pasado a `main`. Se permiten dos modos de operación:

- `approx`: ejecuta los algoritmos aproximados (Greedy, GRASP, Búsqueda Local).
- `exact`: ejecuta el algoritmo de ramificación y poda sobre cada instancia.

Cada modo recorre automáticamente los ficheros en el directorio `data/` y aplica el conjunto de algoritmos sobre ellos, generando salidas tabuladas por consola.

Capítulo 4

Algoritmos Implementados

En esta práctica se han abordado tanto algoritmos heurísticos como un método exacto para resolver el *MDP*. A continuación se describen los enfoques utilizados.

4.1. Algoritmos Aproximados

4.1.1. Fase Constructiva

Greedy (Voraz)

El algoritmo voraz parte de la premisa de construir una solución iteración a iteración, eligiendo en cada una la opción que (en apariencia) maximiza el resultado.

Primero se calcula el centro de gravedad del conjunto completo de puntos. A continuación, se selecciona como primer elemento aquel que esté más alejado de este centro; con esto logramos iniciar la construcción desde el elemento más alejado.

Posteriormente, en cada iteración, se elige el elemento no seleccionado cuya incorporación aumente más la dispersión (medida como la suma de distancias respecto al centroide de los ya seleccionados). El proceso se repite hasta completar el subconjunto de tamaño m. Aunque no garantiza optimalidad, ofrece resultados razonablemente buenos con un tiempo de ejecución muy bajo.

GRASP

El método GRASP (*Greedy Randomized Adaptive Search Procedure*) combina una estrategia voraz con aleatoriedad controlada para generar soluciones más variadas. En lugar de seleccionar siempre el mejor candidato, GRASP construye en cada paso una *Lista Restringida de Candidatos* (LRC), que contiene los elementos con mayor dispersión que se ajusten al tamaño del LRC, y elige aleatoriamente uno de ellos.

Este proceso se repite varias veces. Cada ejecución puede dar lugar a una solución diferente, lo que permite escapar de zonas del espacio de búsqueda donde el algoritmo voraz se podría quedar atrapado. Al final, se conserva la mejor solución encontrada tras ejecutar todas las repeticiones.

El rendimiento de GRASP depende de dos parámetros:

- `|LRC|`: número de elementos en la lista restringida, que controla el equilibrio entre aleatoriedad y voracidad.
- `iterations`: número de repeticiones independientes del proceso de construcción.

4.1.2. Fase de Mejora

Búsqueda Local por Intercambio (1-in 1-out)

Partiendo de un conjunto ya seleccionado, se analiza su vecindario probando intercambios entre un elemento del subconjunto y otro que no esté incluido en él. Para esto se usa una estrategia conocida como "primera mejora.º .ªnsiosa", lo que quiere decir que en cuanto se encuentra un cambio que aumente la diversidad, se acepta de inmediato.

Este procedimiento se repite hasta alcanzar un óptimo local, es decir, hasta que ya no haya forma de seguir mejorando. Esta fase se aplica tanto a la solución generada por el algoritmo Greedy como a la mejor opción que se haya encontrado con GRASP.

4.2. Algoritmo Exacto: Branch and Bound

Se ha implementado un algoritmo de Ramificación y Poda que explora el espacio de soluciones de forma sistemática. Cada nodo representa una solución parcial (conjunto incompleto de elementos seleccionados).

Representación del nodo

Cada nodo (clase `BnBNode`) mantiene:

- Los índices seleccionados hasta el momento.
- Los candidatos restantes.
- El último elemento añadido (para evitar duplicados).
- La cota parcial acumulada (z_1).
- Una cota superior estimada ($z_1 + z_2 + z_3$).

Cálculo de la cota superior

La estimación de la cota superior (*upper bound*) es clave para decidir si un nodo merece ser explorado. Esta cota se construye como la suma de tres componentes:

- z_1: suma de distancias entre los elementos ya seleccionados en el nodo actual.
- z_2: aportación de un candidato individual al conjunto actual (suma de distancias con los seleccionados).
- z_3: estimación de la posible ganancia adicional al combinar ese candidato con otros elementos aún no seleccionados.

Para cada candidato en la lista de elementos no seleccionados, se calcula su z_2 y z_3 de forma individual. La contribución z_3 se obtiene como la suma de las distancias de ese candidato con los elementos restantes con los que forma las combinaciones más diversas, es decir, los más alejados entre sí. Este cálculo se realiza de forma independiente, sin tener en cuenta aún la selección final.

Una vez evaluados todos los candidatos, se suman sus z_2 y z_3 y se seleccionan los mejores según ese total estimado. Con ellos se completa la cota superior, que se suma a z_1, ofreciendo una predicción optimista del valor que se podría alcanzar si se continúa expandiendo ese nodo.

Expansión y poda

El algoritmo emplea un vector ordenado en orden descendente por cota superior. Se generan hijos añadiendo un nuevo candidato al conjunto parcial, y se descartan aquellos cuya cota no supere la mejor solución encontrada hasta el momento.

Se utiliza la solución obtenida mediante Greedy o GRASP como cota inferior inicial, acelerando así la poda de ramas poco prometedoras.

Métricas

Durante la ejecución se contabiliza el número de nodos generados y el tiempo de resolución por instancia, lo que permite comparar el coste computacional con los métodos aproximados.

Capítulo 5

Resultados Experimentales

Con el objetivo de evaluar el comportamiento de los algoritmos desarrollados, se han realizado pruebas sobre diversas instancias del problema con diferentes tamaños (n), dimensiones (k) y tamaños de subconjunto (m). Los experimentos se han centrado en dos ejes principales: la calidad de las soluciones obtenidas (valor z) y el tiempo de ejecución (en milisegundos). En el caso del algoritmo exacto, también se ha registrado el número de nodos generados.

5.1. Instancias evaluadas

Las instancias utilizadas se encuentran en el directorio `data/` y siguen el formato especificado previamente. Incluyen configuraciones con:

- $n \in \{15, 20, 30\}$
- $k \in \{2, 3\}$
- $m \in \{2, 3, 4, 5\}$

Esto proporciona un conjunto amplio y equilibrado de casos para analizar el rendimiento tanto en instancias pequeñas como moderadamente grandes.

5.2. Resultados del algoritmo exacto (Branch and Bound)

El algoritmo de ramificación y poda ha sido ejecutado sobre todas las instancias. En cada caso, devuelve la solución óptima junto con el tiempo necesario para encontrarla y el número total de nodos generados durante la búsqueda.

Se observa que, aunque el método garantiza optimalidad, el número de nodos y el tiempo crecen rápidamente con m, lo que limita su aplicabilidad a instancias de tamaño reducido o moderado. Por ejemplo, para $n = 30$, $k = 3$ y $m = 5$, se requieren más de 2.800 nodos y varios segundos de ejecución.

Cuadro 5.1: Resultados del algoritmo Branch and Bound (Greedy)

Problema	**n**	**K**	**m**	**z**	**S**	**CPU (s)**	**Nodos**
max_div_30_3	30	3	2	13.07	{16, 6}	14.168	0
max_div_30_3	30	3	3	34.28	{5, 16, 23}	184.374	34
max_div_30_3	30	3	4	63.69	{5, 13, 16, 23}	1673.016	454
max_div_30_3	30	3	5	99.58	{5, 13, 14, 16, 23}	8785.564	2891
max_div_30_2	30	2	2	11.66	{8, 27}	28.459	0
max_div_30_2	30	2	3	28.95	{8, 27, 1}	175.898	30
max_div_30_2	30	2	4	52.77	{8, 27, 1, 10}	1559.495	442
max_div_30_2	30	2	5	80.90	{8, 27, 1, 10, 12}	8956.591	3082
max_div_15_2	15	2	2	11.86	{8, 6}	2.197	0
max_div_15_2	15	2	3	27.38	{0, 6, 8}	21.141	36
max_div_15_2	15	2	4	49.84	{0, 5, 6, 8}	136.850	162
max_div_15_2	15	2	5	79.13	{0, 3, 5, 6, 8}	255.649	819
max_div_20_3	20	3	2	11.80	{12, 13}	3.658	0
max_div_20_3	20	3	3	30.87	{12, 13, 7}	38.332	20
max_div_20_3	20	3	4	56.68	{2, 12, 13, 16}	195.732	199
max_div_20_3	20	3	5	92.81	{12, 13, 7, 2, 16}	589.786	770
max_div_15_3	15	3	2	13.27	{11, 8}	1.400	0
max_div_15_3	15	3	3	31.87	{4, 6, 11}	14.745	29
max_div_15_3	15	3	4	59.76	{11, 8, 4, 10}	47.349	120
max_div_15_3	15	3	5	96.10	{3, 4, 8, 11, 13}	158.033	546
max_div_20_2	20	2	2	8.51	{17, 18}	3.649	0
max_div_20_2	20	2	3	21.99	{17, 18, 8}	38.901	21
max_div_20_2	20	2	4	40.00	{1, 2, 8, 18}	211.854	222
max_div_20_2	20	2	5	63.65	{1, 8, 13, 17, 18}	939.133	1248

Cuadro 5.2: Resultados del algoritmo Branch and Bound (GRASP)

Problema	**n**	**K**	**m**	**z**	**S**	**CPU (s)**	**Nodos**
max_div_30_3	30	3	2	13.07	{16, 6}	13.372	0
max_div_30_3	30	3	3	34.28	{5, 16, 23}	199.636	36
max_div_30_3	30	3	4	63.69	{16, 5, 23, 13}	1659.576	448
max_div_30_3	30	3	5	99.58	{5, 13, 14, 16, 23}	9403.287	3170
max_div_30_2	30	2	2	11.66	{27, 8}	13.301	0
max_div_30_2	30	2	3	28.95	{8, 27, 1}	186.053	30
max_div_30_2	30	2	4	52.77	{1, 8, 10, 27}	1641.765	466
max_div_30_2	30	2	5	80.90	{1, 8, 10, 12, 27}	10347.724	3615
max_div_15_2	15	2	2	11.86	{6, 8}	3.418	3
max_div_15_2	15	2	3	27.38	{0, 6, 8}	46.065	106
max_div_15_2	15	2	4	49.84	{0, 5, 6, 8}	66.091	165
max_div_15_2	15	2	5	79.13	{0, 3, 5, 6, 8}	346.616	1038
max_div_20_3	20	3	2	11.80	{12, 13}	10.973	7
max_div_20_3	20	3	3	30.87	{7, 12, 13}	53.437	39
max_div_20_3	20	3	4	56.68	{2, 12, 13, 16}	210.195	199
max_div_20_3	20	3	5	92.81	{2, 7, 12, 13, 16}	1122.461	1472
max_div_15_3	15	3	2	13.27	{8, 11}	3.857	7
max_div_15_3	15	3	3	31.87	{4, 6, 11}	15.851	33
max_div_15_3	15	3	4	59.76	{4, 8, 10, 11}	54.665	125
max_div_15_3	15	3	5	96.10	{3, 4, 8, 11, 13}	204.750	587
max_div_20_2	20	2	2	8.51	{17, 18}	3.670	0
max_div_20_2	20	2	3	21.99	{8, 17, 18}	48.806	32
max_div_20_2	20	2	4	40.00	{1, 2, 8, 18}	211.547	233
max_div_20_2	20	2	5	63.65	{17, 18, 8, 13, 1}	806.754	1089

5.3. Resultados de algoritmos aproximados

Greedy

El algoritmo Greedy ofrece soluciones de forma extremadamente rápida (menos de 0.01 segundos en la mayoría de casos). Sin embargo, su naturaleza determinista puede llevarlo a quedarse atascado en soluciones subóptimas, especialmente en instancias más complejas.

Cuadro 5.3: Resultados del algoritmo Greedy

Problema	**n**	**K**	**m**	**z**	**S**	**CPU (ms)**
				max_div_15_2		
max_div_15_2	15	2	2	11.86	{8, 6}	0.0038
max_div_15_2	15	2	3	25.72	{8, 6, 3}	0.0045
max_div_15_2	15	2	4	48.41	{8, 6, 3, 10}	0.0055
max_div_15_2	15	2	5	73.56	{8, 6, 3, 10, 1}	0.0067
				max_div_20_2		
max_div_20_2	20	2	2	8.51	{17, 18}	0.0042
max_div_20_2	20	2	3	21.99	{17, 18, 8}	0.0053
max_div_20_2	20	2	4	39.56	{17, 18, 8, 2}	0.0067
max_div_20_2	20	2	5	61.22	{17, 18, 8, 2, 12}	0.0084
				max_div_30_2		
max_div_30_2	30	2	2	11.66	{8, 27}	0.0058
max_div_30_2	30	2	3	28.95	{8, 27, 1}	0.0070
max_div_30_2	30	2	4	52.77	{8, 27, 1, 10}	0.0085
max_div_30_2	30	2	5	80.90	{8, 27, 1, 10, 12}	0.0106
				max_div_15_3		
max_div_15_3	15	3	2	13.27	{11, 8}	0.0039
max_div_15_3	15	3	3	30.32	{11, 8, 4}	0.0046
max_div_15_3	15	3	4	59.76	{11, 8, 4, 10}	0.0060
max_div_15_3	15	3	5	94.75	{11, 8, 4, 10, 13}	0.0073
				max_div_20_3		
max_div_20_3	20	3	2	11.80	{12, 13}	0.0048
max_div_20_3	20	3	3	30.87	{12, 13, 7}	0.0059
max_div_20_3	20	3	4	56.52	{12, 13, 7, 2}	0.0071
max_div_20_3	20	3	5	92.81	{12, 13, 7, 2, 16}	0.0088
				max_div_30_3		
max_div_30_3	30	3	2	13.07	{16, 6}	0.0083
max_div_30_3	30	3	3	33.83	{16, 6, 23}	0.0080
max_div_30_3	30	3	4	63.51	{16, 6, 23, 13}	0.0097
max_div_30_3	30	3	5	99.50	{16, 6, 23, 13, 14}	0.0119

Greedy + Búsqueda Local

Al aplicar búsqueda local sobre la solución Greedy, se consigue una mejora significativa en la mayoría de los casos. El incremento de tiempo es modesto, pero se gana en calidad, alcanzando en ocasiones valores óptimos o muy cercanos a los del método exacto.

Cuadro 5.4: Resultados del algoritmo Greedy + Búsqueda Local

Problema	**n**	**K**	**m**	**z**	**S**	**CPU (ms)**
				max_div_15_2		
max_div_15_2	15	2	2	11.86	{8, 6}	0.0096
max_div_15_2	15	2	3	27.38	{8, 6, 0}	0.0136
max_div_15_2	15	2	4	49.84	{8, 6, 0, 5}	0.0235
max_div_15_2	15	2	5	79.13	{0, 6, 3, 5, 8}	0.0337
				max_div_20_2		
max_div_20_2	20	2	2	8.51	{17, 18}	0.0124
max_div_20_2	20	2	3	21.99	{17, 18, 8}	0.0165
max_div_20_2	20	2	4	40.00	{1, 18, 8, 2}	0.0254
max_div_20_2	20	2	5	63.65	{17, 18, 1, 13, 8}	0.0594
				max_div_30_2		
max_div_30_2	30	2	2	11.66	{8, 27}	0.0169
max_div_30_2	30	2	3	28.95	{8, 27, 1}	0.0238
max_div_30_2	30	2	4	52.77	{8, 27, 1, 10}	0.0344
max_div_30_2	30	2	5	80.90	{8, 27, 1, 10, 12}	0.0482
				max_div_15_3		
max_div_15_3	15	3	2	13.27	{11, 8}	0.0100
max_div_15_3	15	3	3	31.87	{11, 6, 4}	0.0191
max_div_15_3	15	3	4	59.76	{11, 8, 4, 10}	0.0173
max_div_15_3	15	3	5	96.10	{11, 8, 4, 3, 13}	0.0286
				max_div_20_3		
max_div_20_3	20	3	2	11.80	{12, 13}	0.0116
max_div_20_3	20	3	3	30.87	{12, 13, 7}	0.0171
max_div_20_3	20	3	4	56.68	{12, 13, 16, 2}	0.0360
max_div_20_3	20	3	5	92.81	{12, 13, 7, 2, 16}	0.0316
				max_div_30_3		
max_div_30_3	30	3	2	13.07	{16, 6}	0.0178
max_div_30_3	30	3	3	34.28	{16, 5, 23}	0.0289
max_div_30_3	30	3	4	63.69	{16, 5, 23, 13}	0.0404
max_div_30_3	30	3	5	99.58	{16, 5, 23, 13, 14}	0.0577

GRASP + Búsqueda Local

GRASP introduce variabilidad en la fase constructiva, generando múltiples soluciones iniciales. Al aplicar búsqueda local sobre las mejores de estas, se obtiene un equilibrio entre calidad y tiempo. Las soluciones tienden a ser cercanas o iguales a las óptimas en muchos casos, y el tiempo de ejecución sigue siendo manejable (del orden de cientos de milisegundos).

La estabilidad del método ha sido evaluada repitiendo los experimentos con distintos tamaños de LRC y número de iteraciones. En general, aumentar el número de iteraciones mejora ligeramente la calidad, aunque con un coste computacional mayor.

Problema	**n**	**K**	**m**	**Iter**	**|LRC|**	**z**	**CPU (s)**
				max_div_15_2			
max_div_15_2	15	2	2	10	2	11.86	0.198838
max_div_15_2	15	2	2	20	2	11.86	0.384060
max_div_15_2	15	2	2	10	3	11.86	0.197175
max_div_15_2	15	2	2	20	3	11.86	0.386294
max_div_15_2	15	2	3	10	2	27.38	0.233624
max_div_15_2	15	2	3	20	2	27.38	0.457480
max_div_15_2	15	2	3	10	3	27.38	0.242932
max_div_15_2	15	2	3	20	3	27.38	0.459914
max_div_15_2	15	2	4	10	2	49.84	0.272107
max_div_15_2	15	2	4	20	2	49.84	0.536730
max_div_15_2	15	2	4	10	3	49.84	0.271456
max_div_15_2	15	2	4	20	3	49.84	0.529437
max_div_15_2	15	2	5	10	2	79.13	0.311603
max_div_15_2	15	2	5	20	2	79.13	0.603798
max_div_15_2	15	2	5	10	3	79.13	0.311292
max_div_15_2	15	2	5	20	3	79.13	0.601003
				max_div_15_3			
max_div_15_3	15	3	2	10	2	13.27	0.200050
max_div_15_3	15	3	2	20	2	13.27	0.398959
max_div_15_3	15	3	2	10	3	13.27	0.197385
max_div_15_3	15	3	2	20	3	13.27	0.388028
max_div_15_3	15	3	3	10	2	31.87	0.236870
max_div_15_3	15	3	3	20	2	31.87	0.468901
max_div_15_3	15	3	3	10	3	30.99	0.238994
max_div_15_3	15	3	3	20	3	31.87	0.464022
max_div_15_3	15	3	4	10	2	59.76	0.276616
max_div_15_3	15	3	4	20	2	59.76	0.539445
max_div_15_3	15	3	4	10	3	59.76	0.275083
max_div_15_3	15	3	4	20	3	59.76	0.541290
max_div_15_3	15	3	5	10	2	96.10	0.317945
max_div_15_3	15	3	5	20	2	96.10	0.613848
max_div_15_3	15	3	5	10	3	96.10	0.331962
max_div_15_3	15	3	5	20	3	96.10	0.664003

Cuadro 5.5 – continuación

Problema	n	K	m	Iter	|LRC|	z	CPU (ms)
max_div_20_2							
max_div_20_2	20	2	2	10	2	8.37	0.228565
max_div_20_2	20	2	2	20	2	8.51	0.441870
max_div_20_2	20	2	2	10	3	8.51	0.224688
max_div_20_2	20	2	2	20	3	8.51	0.443623
max_div_20_2	20	2	3	10	2	21.99	0.276255
max_div_20_2	20	2	3	20	2	21.99	0.561598
max_div_20_2	20	2	3	10	3	21.99	0.277227
max_div_20_2	20	2	3	20	3	21.99	0.545828
max_div_20_2	20	2	4	10	2	40.00	0.331951
max_div_20_2	20	2	4	20	2	40.00	0.642612
max_div_20_2	20	2	4	10	3	40.00	0.331089
max_div_20_2	20	2	4	20	3	40.00	0.669843
max_div_20_2	20	2	5	10	2	63.65	0.387096
max_div_20_2	20	2	5	20	2	63.65	0.730288
max_div_20_2	20	2	5	10	3	63.65	0.390012
max_div_20_2	20	2	5	20	3	63.65	0.749686
max_div_20_3							
max_div_20_3	20	3	2	10	2	11.80	0.232412
max_div_20_3	20	3	2	20	2	11.80	0.451869
max_div_20_3	20	3	2	10	3	11.80	0.249464
max_div_20_3	20	3	2	20	3	11.80	0.478741
max_div_20_3	20	3	3	10	2	30.87	0.282657
max_div_20_3	20	3	3	20	2	30.87	0.564624
max_div_20_3	20	3	3	10	3	30.87	0.283769
max_div_20_3	20	3	3	20	3	30.87	0.582708
max_div_20_3	20	3	4	10	2	56.68	0.364393
max_div_20_3	20	3	4	20	2	56.68	0.663612
max_div_20_3	20	3	4	10	3	56.68	0.336480
max_div_20_3	20	3	4	20	3	56.68	0.659705
max_div_20_3	20	3	5	10	2	92.81	0.417013
max_div_20_3	20	3	5	20	2	92.81	0.750277
max_div_20_3	20	3	5	10	3	92.81	0.388329
max_div_20_3	20	3	5	20	3	92.81	0.758012

Cuadro 5.5 – continuación

Problema	n	K	m	Iter	|LRC|	z	CPU (ms)
max_div_30_2							
max_div_30_2	30	2	2	10	2	11.66	0.271336
max_div_30_2	30	2	2	20	2	11.66	0.537522
max_div_30_2	30	2	2	10	3	11.66	0.264694
max_div_30_2	30	2	2	20	3	11.66	0.526932
max_div_30_2	30	2	3	10	2	28.95	0.341059
max_div_30_2	30	2	3	20	2	28.95	0.677378
max_div_30_2	30	2	3	10	3	28.95	0.337201
max_div_30_2	30	2	3	20	3	28.95	0.660266
max_div_30_2	30	2	4	10	2	52.77	0.413436
max_div_30_2	30	2	4	20	2	52.77	0.795362
max_div_30_2	30	2	4	10	3	52.77	0.420530
max_div_30_2	30	2	4	20	3	52.77	0.857501
max_div_30_2	30	2	5	10	2	80.90	0.495082
max_div_30_2	30	2	5	20	2	80.90	0.938425
max_div_30_2	30	2	5	10	3	80.90	0.503167
max_div_30_2	30	2	5	20	3	80.90	0.954946
max_div_30_3							
max_div_30_3	30	3	2	10	2	13.07	0.363972
max_div_30_3	30	3	2	20	2	13.07	0.669674
max_div_30_3	30	3	2	10	3	12.58	0.381095
max_div_30_3	30	3	2	20	3	13.07	0.680204
max_div_30_3	30	3	3	10	2	34.28	0.441359
max_div_30_3	30	3	3	20	2	34.28	0.859965
max_div_30_3	30	3	3	10	3	34.28	0.445457
max_div_30_3	30	3	3	20	3	34.28	0.841901
max_div_30_3	30	3	4	10	2	63.69	0.503097
max_div_30_3	30	3	4	20	2	63.69	0.896675
max_div_30_3	30	3	4	10	3	63.69	0.446329
max_div_30_3	30	3	4	20	3	63.69	0.815972
max_div_30_3	30	3	5	10	2	99.58	0.514328
max_div_30_3	30	3	5	20	2	99.58	1.127744
max_div_30_3	30	3	5	10	3	99.58	0.527854
max_div_30_3	30	3	5	20	3	99.58	0.997167

5.4. Comparativa global

En conjunto, los resultados obtenidos permiten extraer conclusiones claras sobre el comportamiento de cada enfoque:

- **Branch and Bound** garantiza siempre la solución óptima, lo que lo convierte en la referencia para evaluar la calidad de otros métodos. Sin embargo, su coste computacional crece rápidamente con el tamaño del problema, tanto en tiempo de ejecución como en el número de nodos explorados, lo que lo hace inviable para instancias grandes.
- **Greedy** destaca por su rapidez extrema, siendo prácticamente instantáneo incluso en los casos más complejos. No obstante, esta eficiencia viene acompañada de soluciones de menor calidad, especialmente en instancias con alta complejidad estructural.
- **Greedy + Búsqueda Local** conserva la eficiencia del algoritmo voraz, pero logra mejoras significativas en la calidad de la solución gracias a una fase de mejora. El coste en tiempo sigue siendo muy bajo, por lo que resulta una alternativa eficaz en contextos donde se busca un buen compromiso entre rapidez y rendimiento.
- **GRASP + Búsqueda Local** proporciona soluciones de gran calidad de forma consistente. Su comportamiento se adapta bien a diferentes instancias y puede ajustarse mediante parámetros como el tamaño del conjunto restringido de candidatos (LRC) o el número de iteraciones. Aunque su tiempo de ejecución es mayor que el de las heurísticas anteriores, se mantiene dentro de márgenes aceptables.

Este balance entre calidad, coste computacional y adaptabilidad permite seleccionar el algoritmo más adecuado según los requisitos específicos del problema: necesidad de optimalidad, restricciones de tiempo, o tamaño de la instancia.

Capítulo 6

Conclusión

Este trabajo ha abordado el *Maximum Diversity Problem* (MDP), un problema combinatorio relevante en diversos contextos donde se busca seleccionar subconjuntos dispersos dentro de un conjunto más amplio. A lo largo del desarrollo, se han implementado tanto algoritmos exactos como aproximados, analizando sus resultados sobre un conjunto variado de instancias.

Entre los algoritmos exactos, el enfoque de *Branch and Bound* ha demostrado ser eficaz para obtener soluciones óptimas, aunque con un crecimiento exponencial en coste computacional, lo que limita su aplicabilidad a instancias de tamaño reducido o moderado. Tanto el tiempo como el número de nodos pueden verse notablemente reducidos mediante el uso de una heurística más efectiva para calcular la cota superior. Esto sugiere que una mejora en dicha estimación permitiría al algoritmo exacto podar más ramas del árbol de búsqueda, acelerando significativamente la resolución del problema sin comprometer la optimalidad de los resultados.

En cuanto a los métodos aproximados, se ha comprobado que:

- El algoritmo *Greedy*, aunque extremadamente rápido, no logra siempre soluciones competitivas (eficientes).
- La incorporación de una fase de *Búsqueda Local* tras el Greedy mejora sustancialmente la calidad de los resultados sin apenas incrementar el tiempo de cómputo.
- El método *GRASP + Búsqueda Local* ofrece un excelente compromiso entre calidad y flexibilidad, permitiendo explorar distintas configuraciones mediante parámetros como el número de iteraciones y el tamaño del conjunto LRC.

El análisis experimental ha reforzado estas observaciones, permitiendo comparar no solo los valores de dispersión alcanzados, sino también los tiempos de ejecución y la escalabilidad de cada enfoque. En conjunto, este estudio proporciona una visión completa del equilibrio entre precisión y eficiencia en el contexto del MDP.