UNIVERSIDAD DE GRANADA

Metaheurísticas

Práctica Nº 1

Problema de Mínima Dispersion Diferencial (MDDP)

Curso Académico: **2024/2025**

Horario de prácticas: Jueves 15:30 – 17:30
Email: angelguesan@correo.ugr.es
Nombre: Ángel Sánchez Guerrero
DNI: 77147702K

Índice

1. Descripción y formulación del problema

El Problema de la Mínima Dispersión Diferencial (Minimum Differential Dispersion Problem, MDDP) es un problema de optimización combinatoria NP-completo. Su formulación es aparentemente sencilla, pero su resolución resulta computacionalmente compleja incluso para instancias de tamaño moderado, superando la hora de cómputo para casos de tamaño 50.

El problema consiste en seleccionar un subconjunto M de m elementos de un conjunto inicial S con n elementos (donde $n > m$), de forma que se minimice la dispersión entre los elementos escogidos. Para cada par de elementos, se conoce la distancia entre ellos, representada en una matriz $D = (d_{ij})$ de dimensión $n \times n$.

En el caso específico del MDDP, la medida de dispersión se calcula de la siguiente manera:

1. Para cada elemento v seleccionado, se calcula su valor $\Delta(v)$ como la suma de las distancias de este elemento al resto de elementos seleccionados.
2. La dispersión de una solución, denotada como $diff(S)$, se define como la diferencia entre los valores extremos de Δ:
$$diff(S) = \text{máx}\{\Delta(v) : v \in S\} - \text{mín}\{\Delta(v) : v \in S\}$$
3. El objetivo es minimizar esta medida de dispersión:
$$\text{mín}\, diff(S)$$

Formalmente, el problema se puede formular mediante la siguiente expresión:

$$\text{Minimizar} \ \max_{x_i \in M} \left(\sum_{j \in M} d_{ij} \right) - \min_{x_i \in M} \left(\sum_{j \in M} d_{ij} \right) \ \text{con} \ M \subset S, |M| = m$$

Este problema tiene múltiples aplicaciones prácticas, como:

- Ubicación óptima de instalaciones públicas (como farmacias, hospitales o estaciones de servicio)
- Selección de grupos homogéneos en entornos corporativos o educativos
- Identificación de redes densas en análisis de grafos
- Reparto equitativo de recursos
- Problemas de flujo en redes de transporte o comunicación

Debido a su complejidad computacional, resulta necesario emplear métodos aproximados para su resolución, como algoritmos voraces (greedy), búsqueda local o metaheurísticas más avanzadas, que permitan obtener soluciones de calidad en un tiempo razonable.

2. Breve descripción de la aplicación de los algoritmos empleados al problema

2.1. Representación de soluciones

La representación de soluciones utilizada en este trabajo es un vector de enteros que contiene los índices de los m elementos seleccionados del conjunto original de n elementos. En la implementación, esto se maneja como un objeto `tSolution`, que es simplemente un vector de enteros donde cada posición contiene el índice (entre 0 y n-1) de un elemento seleccionado.

Esta representación garantiza que:

- Cada solución contiene exactamente m elementos
- No hay elementos repetidos
- El orden de los elementos no es relevante, aunque por simplicidad se mantienen ordenados

Por ejemplo, para un problema con $n = 100$ y $m = 10$, una solución válida podría ser el vector:

$$S = [3, 15, 24, 36, 42, 55, 67, 78, 83, 97]$$

2.2. Cálculo de la función objetivo

La función objetivo del MDDP evalúa la dispersión diferencial de una solución dada. Se implementa de la siguiente manera:

Algorithm 1 Cálculo de la dispersión diferencial

```
1:  function FITNESS(solucion)
2:      sumas ← vector de tamaño m inicializado con ceros
3:      for i ← 0 hasta m − 1 do
4:          for j ← 0 hasta m − 1 do
5:              if i ≠ j then
6:                  idx1 ← solucion[i]
7:                  idx2 ← solucion[j]
8:                  sumas[i] ← sumas[i] + distancias[idx1][idx2]
9:              end if
10:         end for
11:     end for
12:     maxSuma ← máximo valor en sumas
13:     minSuma ← mínimo valor en sumas
14:     return maxSuma − minSuma
15: end function
```

2.3. Factorización del cálculo de la función objetivo

Para mejorar la eficiencia, se implementa una factorización que permite calcular el efecto de un cambio en la solución sin recalcular completamente la función objetivo. Esta factorización mantiene las sumas de distancias para cada elemento seleccionado y las actualiza cuando se realiza un movimiento:

Algorithm 2 Cálculo factorizado al reemplazar un elemento

```
1: function FITNESS_FACTORIZADO(solucion, info, pos_cambio, nuevo_valor)
2:     viejo_valor ← solucion[pos_cambio]
3:     nueva_suma ← 0
4:     nuevas_sumas ← copia de info.sumas
5:     for i ← 0 hasta m − 1 do
6:         if i ≠ pos_cambio then
7:             idx ← solucion[i]
8:             nueva_suma ← nueva_suma + distancias[nuevo_valor][idx]
9:             nuevas_sumas[i] ← nuevas_sumas[i] − distancias[viejo_valor][idx] + distancias[nuevo_valor][idx]
10:        end if
11:    end for
12:    nuevas_sumas[pos_cambio] ← nueva_suma
13:    max_suma ← máximo valor en nuevas_sumas
14:    min_suma ← mínimo valor en nuevas_sumas
15:    return max_suma − min_suma
16: end function
```

2.4. Operador de vecindad: Intercambio

El operador de vecindad utilizado en la búsqueda local es el operador de intercambio (Int), que consiste en reemplazar un elemento seleccionado por otro no seleccionado. Este operador mantiene la cardinalidad de la solución constante y garantiza que todas las soluciones vecinas sean factibles.

Algorithm 3 Operador de intercambio

```
1: function INTERCAMBIO(solucion, pos, nuevo_valor)
2:     nueva_solucion ← copia de solucion
3:     nueva_solucion[pos] ← nuevo_valor
4:     return nueva_solucion
5: end function
```

2.5. Generación de soluciones aleatorias

Para inicializar los algoritmos, se generan soluciones aleatorias válidas mediante el siguiente procedimiento:

Algorithm 4 Generación de solución aleatoria

```
1: function CrearSolucionAleatoria
2:     conjunto_unico ← conjunto vacío
3:     while tamaño de conjunto_unico < m do
4:         valor_aleatorio ← entero aleatorio entre 0 y n − 1
5:         Insertar valor_aleatorio en conjunto_unico
6:     end while
7:     solucion ← convertir conjunto_unico a vector
8:     Ordenar solucion
9:     return solucion
10: end function
```

Estas definiciones y operadores comunes serán utilizados por los distintos algoritmos implementados para resolver el problema MDDP, proporcionando una base unificada para su desarrollo y análisis.

3. Descripción en pseudocódigo de los algoritmos implementados

En esta sección se presentan los pseudocódigos específicos de los algoritmos implementados para resolver el problema MDDP. Estos algoritmos utilizan los operadores y definiciones comunes presentados en la sección anterior.

3.1. Algoritmo Aleatorio

El algoritmo aleatorio es el más simple de los implementados. Consiste en generar múltiples soluciones aleatorias y quedarse con la mejor.

Algorithm 5 Algoritmo de Búsqueda Aleatoria

```
1: function OptimizarAleatorio(problema, max_evals)
2:     mejor_solucion ← nulo
3:     mejor_fitness ← ∞
4:     for i ← 1 hasta max_evals do
5:         solucion ← CrearSolucionAleatoria()
6:         fitness ← problema.fitness(solucion)
7:         if fitness < mejor_fitness then
8:             mejor_solucion ← solucion
9:             mejor_fitness ← fitness
10:        end if
11:    end for
12:    return {mejor_solucion, mejor_fitness}
13: end function
```

3.2. Algoritmo Greedy

El algoritmo Greedy para el problema MDDP construye una solución progresivamente, comenzando con un elemento aleatorio y añadiendo en cada paso el elemento que minimice la dispersión de la solución parcial.

Algorithm 6 Algoritmo Greedy para el MDDP

```
1: function OptimizarGreedy(problema, max_evals)
2:     n ← problema.getN()
3:     m ← problema.getM()
4:     evals ← 0
5:     // Inicializar conjuntos de elementos disponibles y seleccionados
6:     disponibles ← {0, 1, 2, ..., n − 1}
7:     solucion ← vector vacío
8:     // Seleccionar primer elemento aleatorio
9:     primer_elemento ← elemento aleatorio de disponibles
10:    Añadir primer_elemento a solucion, eliminar de disponibles
11:    while tamaño de solucion < m Y evals < max_evals do
12:        mejor_dispersion ← ∞
13:        mejor_elemento ← −1
14:        for cada candidato en disponibles do
15:            solucion_temp ← solucion con candidato añadido
16:            dispersion ← problema.fitness(solucion_temp)
17:            evals ← evals + 1
18:            if dispersion < mejor_dispersion then
19:                mejor_dispersion ← dispersion
20:                mejor_elemento ← candidato
21:            end if
22:            if evals ≥ max_evals then break
23:            end if
24:        end for
25:        if evals ≥ max_evals then break
26:        end if
27:        Añadir mejor_elemento a solucion
28:        Eliminar mejor_elemento de disponibles
29:    end while
30:    Ordenar solucion
31:    fitness_final ← problema.fitness(solucion)
32:    return {solucion, fitness_final}
33: end function
```

3.3. Algoritmo de Búsqueda Local

El algoritmo de Búsqueda Local implementado utiliza la estrategia del primer mejor (first improvement), explorando el vecindario según dos modos: aleatorio (randLS) o heurístico (heurLS). Este último prioriza los elementos seleccionados según su contribución al valor de dispersión.

3.3.1. Inicialización y determinación del orden de exploración

Algorithm 7 Algoritmo de Búsqueda Local del Primer Mejor - Parte 1

```
1:  function OptimizarBL(problema, max_evals, modo_exploracion)
2:      n ← problema.getN()
3:      m ← problema.getM()
4:      // Generar solución inicial aleatoria
5:      solucion ← CrearSolucionAleatoria()
6:      fitness ← problema.fitness(solucion)
7:      info ← problema.generarInfoFactorizacion(solucion)
8:      evals ← 1                                   ▷ Contabilizar evaluación inicial
9:      mejora ← verdadero
10:     // Inicializar conjuntos de elementos seleccionados y no seleccionados
11:     seleccionados ← conjunto con elementos de solucion
12:     no_seleccionados ← {0, 1, ..., n − 1} − seleccionados
13:     while mejora Y evals < max_evals do
14:         mejora ← falso
15:         if modo_exploracion = heurLS then
16:             contribuciones ← vector vacío
17:             for i ← 0 hasta m − 1 do
18:                 elem ← solucion[i]
19:                 fitness_sin ← problema.fitness_factorizado(solucion, info, i, elem)
20:                 contribucion ← fitness_sin − fitness
21:                 Añadir (i, contribucion) a contribuciones
22:             end for
23:             Ordenar contribuciones por contribución (menor primero)
24:             posiciones ← extraer índices de posición de contribuciones
25:         else
26:             posiciones ← {0, 1, ..., m − 1}
27:             Barajar posiciones aleatoriamente
28:         end if
29:
```

3.3.2. Exploración del vecindario y selección del primer mejor vecino

Algorithm 8 Algoritmo de Búsqueda Local del Primer Mejor - Parte 2

```
1: function OptimizarBL(problema, max_evals, modo_exploracion) (continuación)
2:     while mejora Y evals < max_evals do
3:         (...código anterior)
4:         for cada pos en posiciones do
5:             if mejora O evals ≥ max_evals then break
6:             end if
7:             elem_actual ← solucion[pos]
8:             candidatos ← vector con elementos de no_seleccionados
9:             Barajar candidatos aleatoriamente
10:            for cada candidato en candidatos do
11:                if evals ≥ max_evals then break
12:                end if
13:                nuevo_fitness ← problema.fitness_factorizado(solucion, info, pos, candidato)
14:                evals ← evals + 1
15:                if nuevo_fitness < fitness then
16:                    // Actualizar conjuntos
17:                    Eliminar elem_actual de seleccionados
18:                    Añadir candidato a seleccionados
19:                    Eliminar candidato de no_seleccionados
20:                    Añadir elem_actual a no_seleccionados
21:                    // Actualizar información factorizada
22:                    problema.actualizarInfoFactorizacion(info, solucion, pos, candidato)
23:                    // Actualizar solución
24:                    solucion[pos] ← candidato
25:                    fitness ← nuevo_fitness
26:                    mejora ← verdadero
27:                    break                                   ▷ Primer mejor encontrado
28:                end if
29:            end for
30:        end for
31:    end while
32:    return {solucion, fitness}
33: end function=0
```

4. Estructura del código y manual de usuario

El proyecto se ha implementado en C++ y estructurado de forma modular para facilitar su comprensión y mantenimiento. A continuación, se describe la organización del código y el proceso para compilarlo y ejecutarlo.

4.1. Estructura de directorios

- **inc/**: Contiene los archivos de cabecera (.h) con las declaraciones de clases y funciones.
  - **mddproblem.h**: Define la clase para el problema MDDP.
  - **randomsearch.h**: Implementa el algoritmo aleatorio.
  - **greedy.h**: Implementa el algoritmo Greedy.
  - **localsearch.h**: Define la búsqueda local y sus variantes.
- **src/**: Contiene los archivos de implementación (.cpp) de las clases y algoritmos.
- **datos_MDD/**: Contiene los 50 ficheros de instancias del problema MDDP con diferentes valores de n y m.
- **build/**: Directorio generado tras la compilación donde se almacenan los ejecutables.
- **common/**: Incluye utilidades comunes para todos los algoritmos:
  - **random.hpp**: Implementación del generador de números aleatorios.
  - **problem.h**: Clase base abstracta para problemas de optimización.
  - **solution.h**: Definición del tipo de solución.
  - **mh.h**: Clase base para metaheurísticas.

4.2. Componentes principales

El proyecto se organiza en las siguientes componentes clave:

- **Representación del Problema**:
  - **mddproblem.h/cpp**: Implementa la clase MDDProblem que encapsula la lógica del problema, incluyendo la carga de instancias desde archivos, el cálculo de la función objetivo y los mecanismos de factorización para optimizar la evaluación de soluciones vecinas.
- **Algoritmos de resolución**:
  - **randomsearch.h/cpp**: Implementa la búsqueda aleatoria que genera soluciones aleatorias hasta agotar el número de evaluaciones.
  - **greedy.h/cpp**: Implementa el algoritmo constructivo Greedy que añade elementos uno a uno seleccionando el mejor candidato en cada paso.
  - **localsearch.h/cpp**: Implementa el algoritmo de búsqueda local con dos modos de exploración del vecindario: aleatorio (randLS) y heurístico (heurLS).
- **Programas ejecutables**:
  - **test_random.cpp**: Programa para probar específicamente el algoritmo aleatorio.
  - **test_greedy.cpp**: Programa para probar específicamente el algoritmo Greedy.
  - **test_localsearch.cpp**: Programa para probar específicamente el algoritmo de búsqueda local.
  - **test_all.cpp**: Programa para probar todos los algoritmos sobre una instancia.

- **run_experiments.cpp**: Programa principal que ejecuta todos los experimentos sobre las 50 instancias del problema, calculando estadísticas y guardando resultados en archivos CSV.

4.3. Compilación del proyecto

El proyecto utiliza CMake como sistema de construcción, lo que facilita su compilación en diferentes plataformas. Para compilar el proyecto, se deben seguir estos pasos:

1. Asegurarse de tener instalado CMake (versión 3.31 o superior) y un compilador de C++ compatible con C++14.
2. Abrir una terminal en el directorio raíz del proyecto.
3. Crear y acceder al directorio de compilación:

   ```
   mkdir -p build
   cd build
   ```

4. Generar los archivos de compilación:

   ```
   cmake ..
   ```

5. Compilar el proyecto:

   ```
   make
   ```

Este proceso generará varios ejecutables en el directorio `build/`, incluyendo `main`, `test_random`, `test_greedy`, `test_localsearch`, `test_all` y `run_experiments`.

4.4. Ejecución de los programas

Una vez compilado el proyecto, se pueden ejecutar los siguientes programas:

- **test_random**: Ejecuta el algoritmo aleatorio sobre una instancia específica.

  ```
  ./test_random <archivo_instancia> [semilla]
  ```

- **test_greedy**: Ejecuta el algoritmo Greedy sobre una instancia específica.

  ```
  ./test_greedy <archivo_instancia> [semilla]
  ```

- **test_localsearch**: Ejecuta ambas versiones del algoritmo de búsqueda local (randLS y heurLS) sobre una instancia y compara sus resultados.

  ```
  ./test_localsearch <archivo_instancia> [semilla]
  ```

- **test_all**: Ejecuta todos los algoritmos sobre una instancia específica para comparar resultados.

  ```
  ./test_all <archivo_instancia> [semilla]
  ```

- **run_experiments**: Ejecuta todos los experimentos sobre el conjunto completo de instancias y genera archivos CSV con los resultados.

  ```
  ./run_experiments <directorio_instancias>
  ```

Este programa utiliza automáticamente las 5 semillas predefinidas (1234, 5678, 9012, 3456, 7890) y genera cuatro archivos de resultados:

- `results_detailed.csv`: Resultados detallados de cada ejecución.
- `results_by_case.csv`: Resultados promediados por caso.
- `results_by_size.csv`: Resultados agrupados por tamaño de instancia.
- `results_global.csv`: Resultados globales por algoritmo.

En todos los programas, si no se especifica una semilla, se utiliza el valor por defecto 42.

4.5. Ejemplo de ejecución

A continuación, se muestra un ejemplo real de ejecución del algoritmo de búsqueda local para la instancia GKD-b_10_n25_m7:

```
$ ./build/test_localsearch datos_MDD/GKD-b_10_n25_m7.txt 222
Ejecutando búsqueda local con exploración ALEATORIA (randLS)...
Fitness (randLS): 47.9688
Evaluaciones: 208
Tiempo de ejecución: 0.000267862 segundos
Elementos seleccionados: 19 12 16 17 5 6 2

Ejecutando búsqueda local con exploración HEURÍSTICA (heurLS)...
Fitness (heurLS): 39.0557
Evaluaciones: 181
Tiempo de ejecución: 0.000349462 segundos
Elementos seleccionados: 5 1 16 17 9 6 12
```

5. Resultados experimentales

En esta sección se presentan los resultados obtenidos de la ejecución de los algoritmos implementados sobre los 50 casos del problema MDDP. Se muestran tres tablas correspondientes a cada uno de los algoritmos: Greedy, Búsqueda Local con exploración aleatoria (LSrandom) y Búsqueda Local con exploración heurística (LSheur).

5.1. Configuración experimental

Para garantizar la reproducibilidad de los resultados y seguir las indicaciones del guión de prácticas, se ha utilizado la siguiente configuración experimental:

- **Semillas utilizadas**: Se han utilizado las siguientes semillas para el generador de números aleatorios:
  - Semilla 1: 1234
  - Semilla 2: 5678
  - Semilla 3: 9012
  - Semilla 4: 3456
  - Semilla 5: 7890

  Estas semillas se utilizan de manera diferente según el algoritmo:
  - **RandomSearch**: Utiliza únicamente la primera semilla (1234).
  - **GreedySearch**: También utiliza solo la primera semilla (1234).
  - **Búsqueda Local (RandLS y HeurLS)**: Utilizan las 5 semillas diferentes para realizar 5 ejecuciones independientes por cada instancia.
- **Número máximo de evaluaciones**: Se estableció un límite de 100000 evaluaciones de la función objetivo.
- **Criterio de parada**: Además del límite de evaluaciones, los algoritmos de búsqueda local se detienen cuando no encuentran mejora en todo el vecindario explorado.
- **Medición de tiempos**: Para medir los tiempos de ejecución se utilizó el cronómetro de alta resolución proporcionado por la biblioteca `<chrono>` de C++.

5.2. Resultados por algoritmo

5.2.1. Resultados obtenidos por el Algoritmo Greedy en el MDD

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0	4.61e-05
GKD-b_2_n25_m2	0	4.74e-05
GKD-b_3_n25_m2	0	4.52e-05
GKD-b_4_n25_m2	0	5.92e-05
GKD-b_5_n25_m2	0	4.60e-05
GKD-b_6_n25_m7	426.36	0.00024
GKD-b_7_n25_m7	189.66	0.00021
GKD-b_8_n25_m7	264.56	0.00021
GKD-b_9_n25_m7	320.03	0.00023
GKD-b_10_n25_m7	359.59	0.00024
GKD-b_11_n50_m5	1994.96	0.00026
GKD-b_12_n50_m5	968.95	0.00025
GKD-b_13_n50_m5	830.04	0.00024
GKD-b_14_n50_m5	1510.09	0.00023
GKD-b_15_n50_m5	1824.35	0.00025
GKD-b_16_n50_m15	218.42	0.00180
GKD-b_17_n50_m15	129.10	0.00188
GKD-b_18_n50_m15	311.69	0.00185
GKD-b_19_n50_m15	240.68	0.00180
GKD-b_20_n50_m15	205.69	0.00181
GKD-b_21_n100_m10	570.81	0.00171
GKD-b_22_n100_m10	365.80	0.00171
GKD-b_23_n100_m10	248.50	0.00179
GKD-b_24_n100_m10	751.02	0.00176
GKD-b_25_n100_m10	235.35	0.00162
GKD-b_26_n100_m30	172.64	0.02036
GKD-b_27_n100_m30	330.81	0.01952
GKD-b_28_n100_m30	511.34	0.02000
GKD-b_29_n100_m30	177.00	0.01959
GKD-b_30_n100_m30	208.07	0.01999
GKD-b_31_n125_m12	493.18	0.00313
GKD-b_32_n125_m12	331.89	0.00290
GKD-b_33_n125_m12	426.97	0.00298
GKD-b_34_n125_m12	285.51	0.00324
GKD-b_35_n125_m12	311.10	0.00294
GKD-b_36_n125_m37	203.14	0.04308
GKD-b_37_n125_m37	73.80	0.04268
GKD-b_38_n125_m37	218.00	0.04352
GKD-b_39_n125_m37	184.82	0.04360
GKD-b_40_n125_m37	118.78	0.04264
GKD-b_41_n150_m15	220.59	0.00589
GKD-b_42_n150_m15	304.22	0.00610
GKD-b_43_n150_m15	597.36	0.00621
GKD-b_44_n150_m15	249.94	0.00619
GKD-b_45_n150_m15	239.29	0.00570
GKD-b_46_n150_m45	138.46	0.08884
GKD-b_47_n150_m45	117.80	0.09024
GKD-b_48_n150_m45	154.71	0.08983
GKD-b_49_n150_m45	206.10	0.08991
GKD-b_50_n150_m45	103.72	0.08941

5.2.2. Resultados obtenidos por el Algoritmo LSrandom en el MDD

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0	5.07e-05
GKD-b_2_n25_m2	0	5.03e-05
GKD-b_3_n25_m2	0	6.36e-05
GKD-b_4_n25_m2	0	5.10e-05
GKD-b_5_n25_m2	0	5.33e-05
GKD-b_6_n25_m7	141.82	0.00042
GKD-b_7_n25_m7	120.75	0.00040
GKD-b_8_n25_m7	93.80	0.00042
GKD-b_9_n25_m7	136.75	0.00036
GKD-b_10_n25_m7	63.26	0.00050
GKD-b_11_n50_m5	425.40	0.00056
GKD-b_12_n50_m5	646.43	0.00038
GKD-b_13_n50_m5	331.75	0.00055
GKD-b_14_n50_m5	724.04	0.00045
GKD-b_15_n50_m5	544.16	0.00037
GKD-b_16_n50_m15	219.92	0.00331
GKD-b_17_n50_m15	88.74	0.00357
GKD-b_18_n50_m15	184.99	0.00250
GKD-b_19_n50_m15	165.19	0.00287
GKD-b_20_n50_m15	205.46	0.00236
GKD-b_21_n100_m10	216.77	0.00318
GKD-b_22_n100_m10	251.99	0.00287
GKD-b_23_n100_m10	240.08	0.00245
GKD-b_24_n100_m10	298.63	0.00258
GKD-b_25_n100_m10	140.58	0.00384
GKD-b_26_n100_m30	126.86	0.02488
GKD-b_27_n100_m30	170.95	0.02750
GKD-b_28_n100_m30	155.70	0.02900
GKD-b_29_n100_m30	133.89	0.02703
GKD-b_30_n100_m30	159.32	0.02426
GKD-b_31_n125_m12	357.26	0.00985
GKD-b_32_n125_m12	131.56	0.00576
GKD-b_33_n125_m12	165.04	0.00779
GKD-b_34_n125_m12	151.84	0.00672
GKD-b_35_n125_m12	187.45	0.00513
GKD-b_36_n125_m37	144.26	0.07322
GKD-b_37_n125_m37	134.53	0.04911
GKD-b_38_n125_m37	191.68	0.04675
GKD-b_39_n125_m37	113.25	0.05203
GKD-b_40_n125_m37	134.20	0.06460
GKD-b_41_n150_m15	172.60	0.01295
GKD-b_42_n150_m15	169.60	0.01040
GKD-b_43_n150_m15	206.97	0.01392
GKD-b_44_n150_m15	162.69	0.01150
GKD-b_45_n150_m15	128.39	0.00935
GKD-b_46_n150_m45	155.02	0.09665
GKD-b_47_n150_m45	96.11	0.10936
GKD-b_48_n150_m45	82.88	0.14974
GKD-b_49_n150_m45	125.81	0.14308
GKD-b_50_n150_m45	111.18	0.12805

5.2.3. Resultados obtenidos por el Algoritmo LSheur en el MDD

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0	5.79e-05
GKD-b_2_n25_m2	0	5.09e-05
GKD-b_3_n25_m2	0	5.31e-05
GKD-b_4_n25_m2	0	5.02e-05
GKD-b_5_n25_m2	0	5.04e-05
GKD-b_6_n25_m7	188.57	0.00040
GKD-b_7_n25_m7	126.04	0.00039
GKD-b_8_n25_m7	115.65	0.00060
GKD-b_9_n25_m7	84.74	0.00075
GKD-b_10_n25_m7	73.24	0.00053
GKD-b_11_n50_m5	763.73	0.00053
GKD-b_12_n50_m5	720.06	0.00057
GKD-b_13_n50_m5	571.87	0.00050
GKD-b_14_n50_m5	811.18	0.00049
GKD-b_15_n50_m5	448.24	0.00041
GKD-b_16_n50_m15	219.09	0.00597
GKD-b_17_n50_m15	73.30	0.00687
GKD-b_18_n50_m15	97.63	0.00885
GKD-b_19_n50_m15	134.20	0.00502
GKD-b_20_n50_m15	109.43	0.00588
GKD-b_21_n100_m10	314.98	0.00476
GKD-b_22_n100_m10	355.05	0.00373
GKD-b_23_n100_m10	161.09	0.00462
GKD-b_24_n100_m10	365.08	0.00376
GKD-b_25_n100_m10	143.21	0.00461
GKD-b_26_n100_m30	125.36	0.07678
GKD-b_27_n100_m30	171.03	0.07891
GKD-b_28_n100_m30	216.22	0.07632
GKD-b_29_n100_m30	96.56	0.07088
GKD-b_30_n100_m30	165.44	0.06138
GKD-b_31_n125_m12	233.92	0.01491
GKD-b_32_n125_m12	180.52	0.00822
GKD-b_33_n125_m12	136.49	0.00879
GKD-b_34_n125_m12	160.29	0.01023
GKD-b_35_n125_m12	186.36	0.01346
GKD-b_36_n125_m37	113.03	0.13308
GKD-b_37_n125_m37	139.07	0.15237
GKD-b_38_n125_m37	140.14	0.15692
GKD-b_39_n125_m37	145.15	0.10897
GKD-b_40_n125_m37	156.33	0.10222
GKD-b_41_n150_m15	185.57	0.02087
GKD-b_42_n150_m15	250.30	0.02327
GKD-b_43_n150_m15	165.72	0.01309
GKD-b_44_n150_m15	235.03	0.02065
GKD-b_45_n150_m15	153.46	0.01660
GKD-b_46_n150_m45	121.80	0.23819
GKD-b_47_n150_m45	85.96	0.25197
GKD-b_48_n150_m45	104.22	0.28723
GKD-b_49_n150_m45	128.89	0.25813
GKD-b_50_n150_m45	151.46	0.24663

5.3. Resultados globales

Una vez analizados los resultados detallados por algoritmo y caso, en esta sección presentamos las tablas resumen que agrupan los datos por tamaño y los resultados globales.

5.3.1. Resultados globales por Tamaño en el MDD

Algoritmo	**Tamaño**	**Desv**	**Tiempo**
Greedy	25	156.02	0.00014
LSrandom	25	55.64	0.00024
LSheur	25	58.83	0.00029
Greedy	50	823.40	0.00104
LSrandom	50	353.61	0.00169
LSheur	50	394.87	0.00351
Greedy	100	357.14	0.01081
LSrandom	100	189.47	0.01476
LSheur	100	211.40	0.03858
Greedy	125	264.72	0.02307
LSrandom	125	171.11	0.03210
LSheur	125	159.13	0.07092
Greedy	150	233.22	0.04783
LSrandom	150	141.12	0.06850
LSheur	150	158.24	0.13766

5.3.2. Resultados globales totales en el MDD

Algoritmo	**Desv**	**Tiempo**
Greedy	366.90	0.01658
LSrandom	182.19	0.02346
LSheur	196.49	0.05019

6. Análisis de resultados

A continuación se presenta un análisis detallado de los resultados obtenidos con los algoritmos implementados para resolver el problema MDDP. Este análisis se centra en interpretar el comportamiento de cada algoritmo y las relaciones entre su diseño y los resultados experimentales.

6.1. Análisis comparativo global

Observando las tablas de resultados globales, podemos extraer varias conclusiones importantes:

- El algoritmo Greedy, aunque es el más rápido (0.01658 segundos en promedio), presenta la mayor desviación respecto a los mejores valores conocidos (366.90 %). Esto es consistente con la naturaleza miope de este algoritmo, que toma decisiones localmente óptimas sin considerar su impacto en la solución final.
- La Búsqueda Local con exploración aleatoria (LSrandom) logra la menor desviación global (182.19 %), con un tiempo computacional moderado (0.02346 segundos). Esto muestra la efectividad de esta metaheurística para escapar de óptimos locales y explorar más ampliamente el espacio de soluciones.
- La Búsqueda Local con exploración heurística (LSheur) obtiene resultados intermedios en calidad (196.49 % de desviación), pero con el mayor coste computacional (0.05019 segundos). Contrario a lo esperado, la versión heurística no supera a la versión aleatoria en términos de calidad global de soluciones.

Este comportamiento podría explicarse por la posible tendencia de la exploración heurística a quedar atrapada en regiones del espacio de búsqueda que, aunque prometedoras inicialmente, no contienen las mejores soluciones globales.

6.2. Análisis por tamaño de instancia

Analizando el comportamiento de los algoritmos según el tamaño de las instancias:

- **Instancias pequeñas (n=25)**: Todos los algoritmos obtienen resultados relativamente buenos, con desviaciones moderadas. Las búsquedas locales (55.64 % y 58.83 %) superan claramente al Greedy (156.02 %), sin una diferencia sustancial entre las dos versiones de BL.
- **Instancias medianas (n=50)**: Se observa una degradación significativa del algoritmo Greedy (823.40 %), mientras que las búsquedas locales mantienen un rendimiento más estable (353.61 % y 394.87 %).
- **Instancias grandes (n=100-150)**: El rendimiento de todos los algoritmos mejora respecto a las instancias medianas, con desviaciones más controladas. Especialmente destacable es que la diferencia entre Greedy y las búsquedas locales se reduce (233.22 % vs. 141.12 % y 158.24 % para n=150).

Esta evolución sugiere que la estructura del problema varía con el tamaño. Particularmente interesante es la mejora relativa del Greedy en instancias grandes, lo que podría indicar que en estas instancias, las decisiones localmente óptimas tienden a producir soluciones globalmente aceptables.

6.3. Análisis del comportamiento de los algoritmos

6.3.1. Algoritmo Greedy

El algoritmo Greedy muestra una alta variabilidad en su rendimiento:

- Excelente en instancias muy pequeñas (n=25, m=2), donde obtiene la solución óptima (desviación 0 %).
- Muy deficiente en instancias medianas, especialmente con n=50, m=5, donde alcanza desviaciones superiores al 1500 %.
- Mejora considerablemente en instancias grandes con valores de m proporcionalmente altos (n=150, m=45), con desviaciones en torno al 100-200 %.

Este comportamiento puede explicarse por la naturaleza combinatoria del problema. En instancias muy pequeñas, el espacio de soluciones es reducido y el algoritmo tiene alta probabilidad de encontrar buenas soluciones. En instancias medianas, la complejidad aumenta rápidamente, pero el algoritmo solo explora un camino. En instancias grandes con m grande, es posible que la distribución de los elementos haga que muchos caminos de construcción lleven a soluciones relativamente buenas.

6.3.2. Búsqueda Local con exploración aleatoria (LSrandom)

Este algoritmo muestra un comportamiento más consistente a través de diferentes tamaños:

- Al igual que Greedy, obtiene soluciones óptimas para instancias muy pequeñas.
- Mantiene desviaciones más controladas en instancias medianas (353.61 % para n=50).
- Consigue las mejores soluciones globales, especialmente efectivas en instancias grandes.

La capacidad de la búsqueda local para mejorar iterativamente una solución inicial permite superar las limitaciones del enfoque Greedy. La exploración aleatoria del vecindario facilita escapar de óptimos locales, lo que explica su mejor rendimiento global.

6.3.3. Búsqueda Local con exploración heurística (LSheur)

Este algoritmo presenta un comportamiento interesante:

- Similar a LSrandom en instancias pequeñas.
- Ligeramente peor que LSrandom en instancias medianas.
- Más efectivo que LSrandom en algunas instancias grandes (especialmente n=125).
- Significativamente más lento que LSrandom en todas las categorías, con tiempos hasta 2-3 veces mayores.

La estrategia heurística para explorar el vecindario parece enfocar la búsqueda demasiado intensamente en ciertas regiones, lo que puede resultar contraproducente si esas regiones no contienen las mejores soluciones. El mayor coste computacional refleja el esfuerzo adicional para ordenar los elementos según su contribución al fitness.

6.4. Relación entre tiempo y calidad

Existe un claro compromiso entre tiempo de ejecución y calidad de las soluciones:

- El algoritmo Greedy es aproximadamente 1.4 veces más rápido que LSrandom, pero obtiene soluciones con una desviación 2 veces mayor.
- LSheur es 2.1 veces más lento que LSrandom, pero no mejora la calidad de las soluciones en promedio.

Estos datos sugieren que LSrandom representa el mejor equilibrio entre calidad y tiempo de ejecución para este problema.

6.5. Conclusiones

Del análisis realizado se pueden extraer las siguientes conclusiones:

1. La Búsqueda Local con exploración aleatoria (LSrandom) es la mejor opción global para resolver el problema MDDP, ofreciendo un buen equilibrio entre calidad de soluciones y tiempo de ejecución.
2. El algoritmo Greedy puede ser una alternativa aceptable cuando el tiempo de ejecución es crítico o para instancias muy grandes con valores de m elevados.
3. La estrategia de exploración heurística (LSheur) no justifica su mayor coste computacional, ya que no mejora significativamente los resultados de la exploración aleatoria.

4. La dificultad del problema no escala linealmente con el tamaño de la instancia, siendo las instancias de tamaño medio (n=50) aparentemente más difíciles que algunas instancias grandes.

5. La factorización de la función objetivo ha sido crucial para la eficiencia de los algoritmos de búsqueda local, permitiendo evaluar movimientos con un coste computacional reducido.

Estos resultados demuestran la efectividad de las técnicas de búsqueda local para problemas de optimización combinatoria como el MDDP, y proporcionan una base sólida para futuras investigaciones con metaheurísticas más avanzadas.