
UNIVERSIDAD DE GRANADA

**Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones**

# Práctica 3. Problema de la Mínima Dispersión Diferencial (MDD)

Doble Grado en Ingeniería Informática y Matemáticas
Metaherísticas, Grupo: A1(Miércoles)

Autor, correo y DNI:

Jaime Corzo Galdó: jaimecrz04@correo.ugr.es 77559045D

26 de mayo de 2025

# Índice

# 1. Descripción del problema

El poblema a resolver es el Problema de Dispersión Diferencial, Minimum Differential Dispersion Problem (MDDP), que es un problema de optimización combinatoria con una formulación sencilla pero una resolución compleja (es NP-completo), que solo con tamaño 50 implica más de 1 hora.
El problema general consiste en seleccionar un subconjunto Sel de $m$ elementos ($|M| = m$) de un conjunto inicial S de n elementos (obviamente, $n > m$) de forma que se minimize la dispersión entre los elementos escogidos.
Además de los $n$ elementos ($e_i, i = 1, ..., n$) y el número de elementos a seleccionar $m$, se dispone de una matriz $D = (d_{ij})$ de dimensión $n * n$ que contiene las distancias entre ellos.

La dispersión de este problema se calcula como sigue:

1. Para cada punto elegido $v$ se calcula $\Delta(v)$ como la suma de las distancias de este punto al resto.
$$\Delta(v) = \sum_{u \in S} d_{uv} \tag{1}$$

2. La dispersión de una solución, denotada como $diff(S)$ se define como la diferencia entre los valores extremos:
$$diff(S) = max_{v \in S}\Delta(v) - min_{v \in S}\Delta(v) \tag{2}$$

3. El objetivo es minimizar dicha medida de dispersión:
$$S^* = min_{S \subseteq V_m} diff(S) \tag{3}$$
donde $S$ es el conjunto solución al problema

# 2. Consideraciones comunes de los algoritmos empleados

En esta práctica vamos a tratar los siguientes algoritmos: Enfriamiento Simulado(ES), Búsqueda Multiarranque Básica (BMB), Búsqueda Local Reiterada (ILS),Hibridación de ILS y ES (ILS_ES), 0.25 puntos y GRASP tanto sin BL como con BL que comentaremos en la siguiente sección. Estos algoritmos buscan, en el caso que la busqueda local se quede en optimos locales, poder seguir generando soluciones nuevas.

Por otro lado, tienen en común una serie de aspectos del código que podemos ver en la carpeta common. Vamos a ir detallandolos poco a poco.

**Notación para los psuedocodigos:**

1. realproblem: copia del parámetro problem que se pasa como argumento en el método optimize
2. m: tamaño de la solución(en formato binario sería el número de unos que tiene la solución)

3. size: tamaño del problema
4. size_poblacion: numero de soluciones que tiene la poblacion
5. info: puede ser un vector o un elemento de la clase MddpFactoringInfo, que implmenta la factorización

## 2.1. Clases solution y util

La clase solution nos sirve para especificar los tipos de datos que más vamos a usar, como son el tipo del fitness, el tipo delos elementos de la solución o el tipo de dato que va a ser la solución, de esta manera, el código es más legible y fácil de mantener.

```
typedef float tFitness;

typedef int tDomain;

typedef int tOption;

typedef tFitness tHeuristic;

typedef std::vector<tDomain> tSolution;
```

Por otro lado, la clase util sobrecarga el operador $\ll$ para un vector,con el objetivo de que nos sea más fácil mostrar el tipo de dato solución, que como acabamos de ver, es un vector.

## 2.2. Clase MH

Tanto BMB, ILS, ILS_ES y GRASP heredan de la clase MH y ES hereda de MHTrayectory, que a su vez hereda de MH, luego sigue teniendo relación con la clase MH la cual tiene un método virtual: optimize, y lo redifinen. Este es el método encargado de, dado un objeto de la clase problem(que ahora analizaremos en que consiste)y un numero máximo de evaluaciones, devuelve un struct ResultMH. Antes de ver que contiene este struct vamos a mostrar el método abstracto optimize:

```
virtual ResultMH optimize(Problem *problem, int maxevals) = 0;
```

Como vemos, devuelve una instancia del struct ResultMH que encapsula la siguiente información:

```
struct ResultMH {
  tSolution solution;
  tFitness fitness;
  unsigned int evaluations;

  ResultMH(tSolution &sol, tFitness fit, unsigned evals)
      : solution(sol), fitness(fit), evaluations(evals) {}
};
```

Esto nos va a facilitar imprimir los resultados de los diferentes algoritmos.

En el caso de ES, al heredar de MHTrayectory, la única diferencia es que el método que implementamos es el siguiente:

```
virtual ResultMH optimize(Problem *problem,
                          const tSolution &current,
                          tFitness fitness, int maxevals) = 0;
```

Esto nos permite que pueda ser llamado desde otros algoritmos pasandole ya una solución creada, que como veremos más adelante, nos será útil para el algoritmo ILS_ES.

## 2.3. Clase mddp

Esta clase hereda de la clase problem, que tiene una serie de métodos abstractos que se implementan en mddp.
mddp encapsula el tamaño n de los datos(size), el tamaño m de la solución(sol_size) y la matriz de distancias. Estos se inicializan mediante los contructores que dispone la clase. Se ha añadido uno para que, dado un archivo de fichero, donde se pasan estos tres datos, lo lea y almacena correctamente cada uno de ellos.
En esta clase además se sobreescribe el método que calcula la función objetivo(método fitness), el método que crea una solución aleatoria(createSolution), algunos getter y los métodos relacionados con la factorización, de la que hablaremos después de comentar estos métodos.

Pseudocódigo función fitness:

**Pseudocódigo función fitness**

```
Funcion fitness(solution) devuelve tFitness
min_fitness ← -1
max_fitness ← 0
Para i desde 0 hasta tamaño de solution - 1 hacer:

      partial_fitness ← calcularDelta(solution, solution[i])

      Si min_fitness es -1 o partial_fitness <min_fitness :

        min_fitness ← partial_fitness

      FinSi

      Si partial_fitness >max_fitness Entonces:

        max_fitness ← partial_fitness

      FinSi

   FinPara

   Devolver max_fitness - min_fitness

FinFuncion
```

Donde calcularDelta es un método también de la clase mddp que dado una solución S y un elemento $v$ de la solución calcula $\Delta(v)$, como se indica en (1). El método fitness por tanto, dado un conjunto solución calcula la dispersión entre los elementos escogidos.

**Pseudocódigo función calcularDelta**

```
Funcion calcularDelta(solution, v) devuelve real
delta ← 0.0

   Para i desde 0 hasta tamaño de solution - 1 hacer:

     delta ← delta + matrix[v][solution[i]]

   FinPara

   Devolver delta

FinFuncion
```

Por último mostramos el pseudocodigo del método createSolution, que genera una

solución aleatoria:

**Pseudocódigo función crearSolución**

```
Función crearSolución()
  conjunto_solución ← conjunto vacío
  Mientras tamaño(conjunto_solución) <tamaño_solución hacer:
    elemento ← número aleatorio entre 0 y tamaño - 1
    Añadir elemento a conjunto_solución
  FinMientras
  Retornar lista creada desde conjunto_solución
FinFunción
```

# 3. Pseudocódigo de cada algoritmo

Como ya hemos indicado, los algoritmos de los que vamos a hablar implementan el método optimize, a continuación vamos a indicar el pseudocódigo de este método para cada uno de estos algoritmos y de las operaciones relevantes de cada uno de ellos.

## 3.1. Algoritmo BMB

La Bl depende mucho de la solución incicial, y además se queda atascado en optimos locales. Con BMB buscamos reducir este problema aplicando BLrandom a distintas soluciones y quedandonos con la mejor. Por ello en vez de llamar a BL con 100000 iteraciones, llamammos 10 veces a BL con 10000 iteraciones cada una. El problema que tiene este algoritmo es que todas las soluciones de las que partimos son aleatorias.

**Pseudocódigo función optimizar (BMB)**

```
Función optimizar(problema, maxevals)
  Asegurar que maxevals >0
  ITERACIONES ← 10
  EVALS_BL ← 10000
  TOTAL_EVALS ← 1
  mejor_solucion ← indefinida
  mejor_fitness ← indefinido
  Para i desde 1 hasta ITERACIONES hacer:
    solucion_inicial ← problema.crearSolucion()
    fitness_inicial ← problema.fitness(solucion_inicial)
    resultado ← BL.optimizar(problema, solucion_inicial,
fitness_inicial, EVALS_BL)
    Si i = 1 Entonces:
      mejor_solucion ← resultado.solucion
      mejor_fitness ← resultado.fitness
    Sino si resultado.fitness <mejor_fitness Entonces:
      mejor_solucion ← resultado.solucion
```

```
      mejor_fitness ← resultado.fitness
    FinSi
  FinPara
  Retornar ResultadoMH(mejor_solucion, mejor_fitness,
TOTAL_EVALS)
FinFunción
```

## 3.2. ILS

Este algoritmo es una mejora del anterior, ya que cada vez que apliquemos la BL, vamos a coger la mejor solución que tengamos, vamos a aplicarle una mutación y a esa le volvemos a aplicar la buśqued local y comparamos si es mejor solución que la anterior.

La mutación se aplica a un 20 % de m, de manera aleatoria, y esos se cambian por otros que no esten en la solucion. Para ello va a a ser necesario llevar una lista de los elementos no seleccionados.

Vamos a comentar primero como funciona la mutación. Para realizarla vamos a generar todas las posibilidades de intercambio entre no seleccionados y seleccionados, los vamos a barajar y vamos a realizar tantos como indique el 20 % de m(SIZE_MUTATION). El pseudocódigo quedaria asi:

**Pseudocódigo función mutación (ILS)**

```
Función mutación(solución_inicial, no_seleccionados,
SIZE_MUTATION)
  Crear lista de listas 'posibilidades'
  Para i desde 0 hasta tamaño(solución_inicial) - 1 hacer:
    aux ← lista vacía
    Para j desde 0 hasta tamaño(no_seleccionados) - 1 hacer:
      Añadir (i, j) a aux
    FinPara
    Barajar aux aleatoriamente
    Añadir aux a posibilidades
  FinPara
  Barajar posibilidades aleatoriamente
  cont ← 0
  ok ← verdadero
  Para i desde 0 hasta tamaño(posibilidades) - 1 y ok hacer:
    Para j desde 0 hasta tamaño(posibilidades[i]) - 1 y ok
hacer:
      pos_out ← posibilidades[i][j].primero
      pos_in ← posibilidades[i][j].segundo
      elem_out ← solución_inicial[pos_out]
      elem_in ← no_seleccionados[pos_in]
      solución_inicial[pos_out] ← elem_in
```

```
      no_seleccionados[pos_in] ← elem_out
      cont ← cont + 1
      Si cont ≥ SIZE_MUTATION Entonces:
        ok ← falso
      FinSi
    FinPara
  FinPara
FinFunción
```

Así, el método optimize nos quedaría así:

**Pseudocódigo función optimizar (ILS)**

```
Función optimizar(problema, maxevals)
  Asegurar que maxevals >0
  rango ← problema.getSolutionDomainRange()
  size ← rango.segundo + 1
  m ← problema.getSolutionSize()
  ITERACIONES ← 9
  EVALS_BL ← 10000
  TOTAL_EVALS ← 1
  MUTATION ← 0.2
  SIZE_MUTATION ← MUTATION * m
  Si SIZE_MUTATION <2 Entonces:
    SIZE_MUTATION ← 2
  FinSi
  solucion_inicial ← problema.crearSolucion()
  fitness_inicial ← problema.fitness(solucion_inicial)
  resultado ← BL.optimizar(problema, solucion_inicial,
fitness_inicial, EVALS_BL)
  mejor_solucion ← resultado.solucion
  mejor_fitness ← resultado.fitness
  Crear vector aux_seleccionados de tamaño size con ceros
  Para i desde 0 hasta m - 1 hacer:
    aux_seleccionados[mejor_solucion[i]] ← 1
  FinPara
  no_seleccionados ← elementos donde aux_seleccionados[i] = 0
  Para i desde 1 hasta ITERACIONES hacer:
    mutación(solucion_inicial, no_seleccionados, SIZE_MUTATION)
    fitness_inicial ← problema.fitness(solucion_inicial)
    resultado ← BL.optimizar(problema, solucion_inicial,
fitness_inicial, EVALS_BL)
    Si resultado.fitness <mejor_fitness Entonces:
      mejor_solucion ← resultado.solucion
      mejor_fitness ← resultado.fitness
    FinSi
    solucion_inicial ← mejor_solucion
```

```
    fitness_inicial ← mejor_fitness
  FinPara
  Retornar ResultadoMH(mejor_solucion, mejor_fitness,
TOTAL_EVALS)
FinFunción
```

## 3.3. ES

Este algoritmo es distinto a los anteriores, en vez de usar la búsqueda local vamos a usar enfriamiento simulado. La idea es intentar evitar el problema del optimo local permitiendo que la solución pueda empeorar, teniendo en cuenta que si permito que empeore mucho es un algoritmo aleatorio, si no empeora nunca es la BL pura pero con solo un intercambio como mucho.

La diferencia radica en que si la nueva solucion que genero es mejor que la anterior me la quedo, si es peor hay veces que me la quedo. Hay que tener en cuenta que sobre todo voy a dejar que las soluciones empeoren al principio del algoritmo, ya que tengo más margen de maniaobra. Si me quedan pocas iteraciones, empeorar a ultima hora en general no ayuda. Evidentemente va a depender tambien de cuanto se empeora.

Para ello vamos a usar una variable T o temperatura. Nos indica cuanto podemos empeorar, inicia en una temperatura inicial fijada y va disminuyendo conforme aumentan las iteraciones hasta llegar como mucho a una temperatura final fijada también. **Siempre se tiene que cumplir que t_inicial > t_final**.

La temepratura se va a actualizar cuando haga muchos cambios(nvecinos $\geq$ max_vecinos) o actualicen muchas soluciones(nexitos $\geq$ max_exitos).

Respecto a la duración, el enfriamiento simulado no puede tardar mas q la BL.

Para reducir la temperatura, vamos a emplear el esquema de Cauchy modificado, explicado en el guión de la práctica. Por último comentar que, como ya indicamos, esta clase hereda de la clase MHTrayectory, por lo que el método optimize que implementa recibe una solución y un fitness a los que aplica el enfriamiento simulado.

**Pseudocódigo función vecindario_operador (ES)**

```
Función vecindario_operador(solucion_inicial, size, m)
  new_sol ← copia de solucion_inicial
  Crear vector aux_seleccionados de tamaño size con ceros
  Para i desde 0 hasta m - 1 hacer:
    aux_seleccionados[solucion_inicial[i]] ← 1
  FinPara
  no_seleccionados ← elementos donde aux_seleccionados[i] = 0
  pos_out ← número aleatorio entre 0 y m - 1
  pos_in ← número aleatorio entre 0 y size - m - 1
  elem_in ← no_seleccionados[pos_in]
  new_sol[pos_out] ← elem_in
```

```
  Retornar new_sol
FinFunción
```

El método optimize quedaría así:

**Pseudocódigo función optimizar (ES)**

```
Función optimizar(problema, solucion_actual, fitness, maxevals)
  Asegurar que maxevals >0
  solucion_inicial ← solucion_actual
  fitness_inicial ← fitness
  u ← 0.2
  phi ← 0.3
  ln_phi ← log(phi)
  t_inicial ← (u * fitness_inicial) / -ln_phi
  t_final ← 1e-3
  Si t_final ≥ t_inicial Entonces:
    t_inicial ← t_final * 10
  FinSi
  temperature ← t_inicial
  rango ← problema.getSolutionDomainRange()
  size ← rango.segundo + 1
  m ← problema.getSolutionSize()
  max_vecinos ← 100 * m
  max_exitos ← 0.1 * max_vecinos
  M ← maxevals / max_vecinos
  beta ← (t_inicial - t_final) / (M * t_inicial * t_final)
  mejor_solucion ← solucion_inicial
  mejor_fitness ← fitness_inicial
  evaluaciones ← 0
  hay_exitos ← verdadero
  Mientras evaluaciones <maxevals y hay_exitos hacer:
    nexitos ← 0
    nvecinos ← 0
    Mientras nexitos <max_exitos y nvecinos <max_vecinos y
evaluaciones <maxevals hacer:
      new_sol ← vecindario_operador(solucion_inicial, size, m)
      new_fitness ← problema.fitness(new_sol)
      delta ← new_fitness - fitness_inicial
      nvecinos ← nvecinos + 1
      evaluaciones ← evaluaciones + 1
      Si delta <0 o Random(0,1) ≤ exp(-delta / temperature)
Entonces:
        solucion_inicial ← new_sol
        fitness_inicial ← new_fitness
        nexitos ← nexitos + 1
        Si fitness_inicial <mejor_fitness Entonces:
```

```
          mejor_solucion ← solucion_inicial
          mejor_fitness ← fitness_inicial
        FinSi
      FinSi
    FinMientras
    temperature ← temperature / (1 + (beta * temperature))
    Si nexitos = 0 Entonces:
      hay_exitos ← falso
    FinSi
  FinMientras
  Retornar ResultMH(mejor_solucion, mejor_fitness, evaluaciones)
FinFunción
```

## 3.4. ILS_ES

Este algoritmo es idéntico a ILS, pero en vez de llamar a BL llamamos a ES. Para ello tenemos objeto de la clase ES como atributo privado, para poder usar su método optimize sobre las soluciones, como vemos en el pseudócodigo del algoritmo(el metodo de mutación es idéntico al de ILS):

**Pseudocódigo función optimizar (ILS_ES)**

```
Función optimizar(problema, maxevals)
  Asegurar que maxevals >0
  size ← tamaño del dominio de la solución
  m ← tamaño de la solución
  ITERATIONS ← 9
  EVALS_ES ← 10000
  TOTAL_EVALS ← 1
  MUTATION ← 0.2
  SIZE_MUTATION ← MUTATION * m
  Si SIZE_MUTATION <2 entonces:
    SIZE_MUTATION ← 2
  FinSi
  solucion_inicial ← problema.crearSolucion()
  fitness_inicial ← problema.fitness(solucion_inicial)
  resultado ← enfriamiento_simulado.optimizar(problema,
solucion_inicial, fitness_inicial, EVALS_ES)
  mejor_solucion ← resultado.solucion
  mejor_fitness ← resultado.fitness
  Inicializar vector aux_seleccionados con ceros de tamaño size
  Para i desde 0 hasta m-1 hacer:
    aux_seleccionados[mejor_solucion[i]] ← 1
  FinPara
  Construir no_seleccionados con los valores no presentes en
mejor_solucion
```

```
  Para i desde 1 hasta ITERATIONS hacer:
    mutacion(solucion_inicial, no_seleccionados, SIZE_MUTATION)
    fitness_inicial ← problema.fitness(solucion_inicial)
    resultado ← enfriamiento_simulado.optimizar(problema,
solucion_inicial, fitness_inicial, EVALS_ES)
    Si resultado.fitness <mejor_fitness entonces:
      mejor_solucion ← resultado.solucion
      mejor_fitness ← resultado.fitness
    FinSi
    solucion_inicial ← mejor_solucion
    fitness_inicial ← mejor_fitness
  FinPara
  Retornar ResultMH(mejor_solucion, mejor_fitness, TOTAL_EVALS)
FinFunción
```

## 3.5. GRASP

**Importante:** Para el algoritmo GRASP no se ha podido seguir al pie de la letra la API debido al uso del greedy aleatorio, ya sabemos que el greedy realiza un dynamic cast sobre Problem para obtener un objeto de tipo Mddp, y poder tener acceso a la madtriz de distancias.

```
1 Mddp *realproblem = dynamic_cast<Mddp *>(problem);
```

Este algoritmo es similar al BMB pero en vez de generar soluciones aleatorias aplicamos la heuristica de usar primero el greedy aleatorizado.

Para ello llevamos una lista de candidatos(LC), con todas las posiciones que no se eencuentran en la solución, y vamos a crear la lista restringida de candidatos(LRC) metiendo aquellos candidatos que sean lo "suficientemente buenos"de la siguiente manera:

1. La LRC es de tamaño variable.
2. Incluye todos los elementos no seleccionados cuya distancia acumulada a los actualmente seleccionados es mayor o igual que el umbral de calidad $\mu = d_{min} + \alpha(d_{max} - d_{min})$, donde $d_{min}$ es la menor distancia coste de los candidatos de LC y $d_{max}$ la mayor distancia.
3. Si el $\alpha$ es muy pequeño no meto casi ninguno y conforme aumenta van entrando más candidatos.
4. Se escoge aleatoriamente un elemento candidato de la LRC y se añade a la solución parcial.
5. En cada nuevo paso del algoritmo hay que actualizar la LC, eliminando los elementos seleccionados en el paso anterior, y construir la nueva LRC recalculando las distancias para los candidatos factibles restantes y aplicando el umbral para filtrar los candidatos a emplear.
6. Repetimos hasta que el tamaño de la solución sea m.

De esta manera obtendriamos una solución a partir del greedy aleatorizado al que luego le aplicamos la búsqueda local, así durante 10 iteraciones y nos quedamos con la mejor solución.

Por otro lado, el Grasp sin BL, es el mismo algoritmo pero sin aplicar en cada paso la búsqueda local a las soluciones del greedy aleatorizado, con el objetivo de comprobar si la heuristica del Greedy es mejor que aplicar hacer BL.

**Pseudocódigo función greedy_aleatorizado (GRASP)**

```
Función greedy_aleatorizado(problema, size, m)
  alpha ← 0.2
  sol_greedy ← solución vacía
  LRC ← lista vacía
  distancias_candidatos ← lista vacía
  lista_candidatos ← [0, 1, ..., size - 1]
  Si m == 1:
    Añadir lista_candidatos[0] a sol_greedy
  Sino si m == 2:
    Añadir lista_candidatos[0] y lista_candidatos[1] a
sol_greedy
  Sino:
    Seleccionar aleatoriamente un elemento de lista_candidatos → 
elem
    Añadir elem a sol_greedy y eliminarlo de lista_candidatos
    Repetir para otro elemento aleatorio → elem
    Añadir elem a sol_greedy y eliminarlo de lista_candidatos
    m ← m - 2
    Mientras m >0:
      Para cada candidato en lista_candidatos:
        Calcular delta respecto a sol_greedy y guardar en
distancias_candidatos
      min_distancia ← mínimo de distancias_candidatos
      max_distancia ← máximo de distancias_candidatos
      umbral ← min_distancia + alpha * (max_distancia -
min_distancia)
      Para cada i en distancias_candidatos:
        Si distancias_candidatos[i] ≤ umbral:
          Añadir lista_candidatos[i] a LRC
      index_in ← posición aleatoria en LRC
      Añadir LRC[index_in] a sol_greedy
      Eliminarlo de lista_candidatos
      Limpiar LRC y distancias_candidatos
      m ← m - 1
  Retornar sol_greedy
FinFunción
```

Con lo que el método optimize queda así:

**Pseudocódigo función optimize (GRASP)**

```
Función optimize(problema, maxevals)
  Verificar que maxevals >0
  realproblem ← conversión de tipo de problema a Mddp
  ITERATIONS ← 10
  EVALS_BL ← 10000
  TOTAL_EVALS ← 1
  Obtener size y m del problema
  Para i desde 1 hasta ITERATIONS:
    solucion_inicial ← greedy_aleatorizado(realproblem, size, m)
    fitness_inicial ← realproblem→fitness(solucion_inicial)
    Si hay_bl:
      result ← BL.optimize(realproblem, solucion_inicial,
fitness_inicial, EVALS_BL)
      Si i == 1 o result.fitness <mejor_fitness:
        mejor_solucion ← result.solution
        mejor_fitness ← result.fitness
    Sino:
      Si i == 1 o fitness_inicial <mejor_fitness:
        mejor_solucion ← solucion_inicial
        mejor_fitness ← fitness_inicial
  Retornar ResultMH(mejor_solucion, mejor_fitness, TOTAL_EVALS)
FinFunción
```

# 4. Estructura del código de la práctica

Esta praćtica se divide en las siguientes carpetas:

1. common: En esta carpeta se encuentran las clases y archivos que son comunes a cualquier problema, y a partir de las cuales debes implmentar tu problema concreto. Tenemos los siguientes archivos:
   - mh.h: Comentado en la sección 2
   - mhtrayectory.h: Hereda de la clase MH e implementa un metodo optimize donde se le pasa como argumento el problema, el número máximo de evaluaciones, y además una solución y su fitness para aplicarle el método optimize.
   - problem.h: Comentado en la sección 2
   - random.h: Para generación de números aleatorios en C++
   - solution.h: Comentado en la sección 2
   - util.h: Comentado en la sección 2
2. data: Los 50 casos considerados de MDPLIB(son archivos de texto) para crear las instancias de la clase mddp. Indican el tamaño del problema(n), el tamaño de la solución(m) y la matriz de distancias

3. inc: Los .h de los distintos algoritmos con los que vamos a trabajar, además del archivo mddp que hereda de la clase problema

4. src: Los .cpp de los distintos algoritmos con los que vamos a trabajar, además del archivo mddp que hereda de la clase problema

## 4.1. Compilar y ejecutar

Disponemos de un cmake para compilar la práctica, que se usa del siguiente modo(tenemos que estar situados en la carpeta donde se encuentra el archivo CMakeLists.txt)

Nota: Al ejecutar el main, si no se le pasa ningún argumento, se toman las semillas que hay definidas en el main, para usar unas semillas distintas se deben pasar como argumento(hay que pasar cinco).

A continuación se muestra un ejemplo de ejecución pasando cinco semillas como argumento, no se muestra nada por pantalla porque los resultados se redirigen a un archivo .csv

Se ha cortado la ejecución con Ctrl+C ya que el objetivo era simplemente mostrar como se realizaba.

```
masInteligentes/Metaheuristicas/P1-ProblemaMinimaDispersion$ rm CMakeCache.txt
jaimecrz3@ubuntujaimecg:~/Documentos/DGIIMGITHUB/DGIIM/Mención-ComputacionYSiste
masInteligentes/Metaheuristicas/P1-ProblemaMinimaDispersion$ cmake .
-- The CXX compiler identification is GNU 13.3.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.3s)
-- Generating done (0.0s)
-- Build files have been written to: /home/usuario/Documentos/DGIIMGITHUB/DGIIM/Mención-ComputacionYSistemasInteligentes/Metaheuristicas/P1-ProblemaMinimaDispersion
jaimecrz3@ubuntujaimecg:~/Documentos/DGIIMGITHUB/DGIIM/Mención-ComputacionYSiste
masInteligentes/Metaheuristicas/P1-ProblemaMinimaDispersion$ make
[  9%] Building CXX object CMakeFiles/main.dir/main.cpp.o
[ 18%] Building CXX object CMakeFiles/main.dir/src/AGE.cpp.o
[ 27%] Building CXX object CMakeFiles/main.dir/src/AGG.cpp.o
[ 36%] Building CXX object CMakeFiles/main.dir/src/AM.cpp.o
[ 45%] Building CXX object CMakeFiles/main.dir/src/BLheur.cpp.o
[ 54%] Building CXX object CMakeFiles/main.dir/src/BLrandom.cpp.o
[ 63%] Building CXX object CMakeFiles/main.dir/src/brutesearch.cpp.o
[ 72%] Building CXX object CMakeFiles/main.dir/src/greedy.cpp.o
[ 81%] Building CXX object CMakeFiles/main.dir/src/mddp.cpp.o
[ 90%] Building CXX object CMakeFiles/main.dir/src/randomsearch.cpp.o
[100%] Linking CXX executable main
[100%] Built target main
jaimecrz3@ubuntujaimecg:~/Documentos/DGIIMGITHUB/DGIIM/Mención-ComputacionYSiste
masInteligentes/Metaheuristicas/P1-ProblemaMinimaDispersion$ ./main 42, 123, 1, 24, 98
^C
jaimecrz3@ubuntujaimecg:~/Documentos/DGIIMGITHUB/DGIIM/Mención-ComputacionYSistemasInteligentes/Metaheuristicas/P1-ProblemaMinimaDispersion$
```

Figura 1: Manera de compilar y ejecutar la práctica

# 5. Experimentos y análisis de resultados

## 5.1. Casos del problema empleados

En esta práctica se han usado 50 casos seleccionados de varios de los conjuntos de instancias disponibles en la MDPLIB (https://grafo.etsii.urjc.es/optsicom/mdp/ ), todas pertenecientes al grupo GDK con distancias aleatorias: 50 del grupo GDK-b con distancias reales con, n entre 25, 50, 75, 100, 125, 150, y m entre 2 y 45 (GDK-bGKD-b_1_n25_m2.txt a GKD-b_50_n150_m45.txt).

Cada uno de los siete algoritmos ya comentados se ha ejecutado pasandole como parámetro el problema(instancia de la clase mddp), y el número máximo de evaluaciones. Esto se ha hecho para cada uno de los 50 ficheros manteniendo siempre $max\_evals = 100000$.

Esto se repite para cada una de las siguientes semillas(se pueden indicar cinco diferentes pasandolas como argumentos en la ejecución del main): 42, 123, 1, 24 y 98

## 5.2. Resultados obtenidos y análisis de los mismos

A continuación se muestran las tablas de la media de los resultados de tiempo y calidad para cada uno de los algoritmos de esta práctica además del Random, Greedy, BLRandom y el mejor algoritmo de la práctica 2, el AM-(10, 0.1), explicado en la práctica 2. Primero vamos a agrupar por tamaño del problema y posteriormente vamos obtener otra en la que se muestre los resultados para todos los tamaños.

| **Algoritmo** | **Tamaño** | **Desv** | **Tiempo** |
|---|---|---|---|
| Random | 50 | 60,63 | 85,6 |
| Greedy | 50 | 81,62 | 0 |
| LSrandom | 50 | 70,5 | 2,9 |
| AM-(10,0.1) | 50 | 50,8 | 329 |
| BMB | 50 | 55,27 | 2,8 |
| ILS | 50 | 46,57 | 1,6 |
| ES | 50 | 67,56 | 14,5 |
| ILS_ES | 50 | 51,37 | 14 |
| GRASP-SIBL | 50 | 53,99 | 2,2 |
| GRASP-NOBL | 50 | 88,96 | 0 |
| Random | 100 | 75,89 | 197 |
| Greedy | 100 | 74,48 | 3,8 |
| LSrandom | 100 | 62,35 | 29,5 |
| AM-(10,0.1) | 100 | 35,61 | 838 |
| BMB | 100 | 44,06 | 31,6 |
| ILS | 100 | 43,23 | 13,7 |
| ES | 100 | 62,6 | 24,5 |
| ILS_ES | 100 | 43,82 | 63,4 |
| GRASP-SIBL | 100 | 42,94 | 25 |
| GRASP-NOBL | 100 | 87,6 | 0 |
| Random | 150 | 78,41 | 329 |
| Greedy | 150 | 69,15 | 22 |
| LSrandom | 150 | 56,06 | 94,6 |
| AM-(10,0.1) | 150 | 36,37 | 1920 |
| BMB | 150 | 40,31 | 91,7 |
| ILS | 150 | 35,87 | 48,8 |
| ES | 150 | 61,24 | 44,5 |
| ILS_ES | 150 | 43,31 | 124 |
| GRASP-SIBL | 150 | 39,22 | 87,3 |
| GRASP-NOBL | 150 | 86,27 | 0,5 |

Cuadro 1: Resultados de los algoritmos Random, Greedy, LSrandom, AM-(10,0.1), BMB, ILS, ES, ILS-ES, GRASP-SIBL y GRASP-NOBL para los tamaños 50, 100 y 150.

| Algoritmo | Desv | Tiempo |
|---|---|---|
| Random | 61,28 | 182 |
| Greedy | 67,06 | 7 |
| LSrandom | 55,71 | 35,3 |
| AM-(10,0.1) | 33,78 | 897 |
| BMB | 38,96 | 35,9 |
| ILS | 35,14 | 18,2 |
| ES | 55,91 | 24,5 |
| ILS_ES | 38,76 | 58,1 |
| GRASP-SIBL | 38,59 | 32,2 |
| GRASP-NOBL | 78,35 | 0,2 |

Cuadro 2: Media de Desviación y Tiempo para los algoritmos Random, Greedy, LSrandom, AM-(10,0.1), BMB, ILS, ES, ILS-ES, GRASP-SIBL y GRASP-NOBL.

Estas tablas muestran la desviación de cada algoritmo respecto a la mejor solución y el tiempo que tardan para cada uno de los casos del problema en milisegundos. Para ayudarnos a realizar el análisis, vamos a ir respondiendo a las siguientes cuestones:

**1. Comparación Blrandom con los nuevos algoritmos que usan la Blrandom**

El objetivo de los nuevos algoritmos de esta práctica, i.e. de las técnicas basadas en trayectorias, es reducir los efectos de las dificultades que puede tener la búsqueda local al llegar a un óptimo local. Vamos a ir comparandolos uno a uno:

- BMB: Este algoritmo mejora notablemente a la búsqueda local, a pesar de que lo "único"que hacemos es crear 10 soluciones aleatorias y aplicarle la BLRandom, esto se explica ya que para este problema la búsqueda local se iba rapidamente a óptimos locales, entonces a pesar de tener 100000 evaluaciones, nunca llegaba a ese número, ahora seguimos tenendo 100000 evaluaciones, por eso el tiempo de ambos algoritmos es similar, pero repartida en 10 iteraciones donde se llama a la búsqueda local con 10000 iteraciones, por lo que podemos generar distintas soluciones y quedarnos con la mejor.
- ILS: Este algoritmo es una mejora del BMB, como podemos ver mejora un poco las desviaciones y en tiempo se reduce también, ya que en vez de en cda iteración trabajar con una solución aleatoria y aplicarle la buśqueda local, trabajamos con la mejor solución hasta el momento, aplicandole unos pocos cambios, luego siempre mejoramos a partir de nuestra mejor solución, no de una completamente aleatoria. Que se reduzca en tiempo nos muestra que generar una nueva solución es más costoso que aplicar un mutación al 20 % de la solución.
- GRASP-SIBL: El objetivo de este algoritmo es, al igual que ILS, mejorar BMB, en este caso, en vez de aplicar la búsqueda local a una solución aleatoria, se la vamos a aplicar a una solución obtenida con una heurística, en este caso el greedy aleatorizado, explicado en el apartado 3.5. Como podemos ver, se obtienen unos resultados similares a los obtenidos con BMB, lo cual tiene sentido ya que como ya sabemos de la práctica 1, el greedy en este problema

no obtiene muy buenos resultados y además el greedy aleatorizado contiene una componente random que hace que en general sea peor que el greedy.

Como podemos ver, estos algoritmos que usan la BLRandom, la mejoran todos ya que minimizan la pérdida de generación de nuevas soluciones por llegar a un óptimo local.

**2. ¿Qué Grasp es mejor?**

Con los algoritmos GRASP-SIBL y GRASP-NOBL, queríamos comprobar como de buena es la heurística del greedy aleatorizado sin aplicar en ningún caso la BLRandom, como hacemos en GRASP-NOBL. Como podemos ver en la tabla de resultados, la desviación aumenta casi al doble aunque como es de esperar el tiempo que tarda de media GRASP-NOBL es casi cero.

Estos resultados nos muestran lo que ya sabiamos, que el Greedy en este problema no obtiene buenos resultados, y si en vez de coger la mejor opción en cada momento, que es la filosofía del Greedy, cogemos uno de manera aleatoria, aunque sea entre una lista restringda de candidatos, es normal que empeore.

Por lo que el GRASP-SIBL, al tener la Búsqueda Local, hace que mejore de una manera notable.

**3. Comparación entre Greedy y Grasp**

Como ya hemos comentado, el algoritmo Grasp usa el Greedy aleatorizado para generar nuevas soluciones, que para este problema ya hemos visto que no obtiene muy buenos resultados, por lo que comparando el Greedy con el Grasp, podemos decir que lo que hace que este último obtenga mejores resultados, es que a cada solución obtenida con el Greedy aleatorizado se le aplicala búsqueda local, si no fuera por ella los resultados serían incluso peores que el Greedy, ya que no estamos aplicando el Greedy directamente, si no una heurística del mismo.

**4. Comentario ES**

El algoritmo de enframiento simulado es en cierta manera a los otros de esta práctica, ya que en vez de usar la BL, hacemos modificaciones muy pequeñas, solo cambiamos un elemento de la solución a lo sumo, y lo más importante, permitimos que de vez en cuando, y de manera aleatoria, la solución empeore, con el objetivo de no quedarnos en óptimos locales.

Evidentemente, al inicio vamos a permitir que empeore con más frecuencia ya que tenemos más margen de mejora que si quedan unas pocas iteraciones. Podemos ver que los resultados obtenidos con la BLRandom son similares a los obtenidos con el ES, eso se puede exlicar por las siguientes razones:

1. En cadaiteración solo se cambia lo umo una posición de la solución, por lo que limita la posibilidad de obtener soluciones más diferentes y con un mayor potencial.

2. Poca diferencia entre temperatura inicial y final, lo que puede hacer que el enfriamiento sea muy corto. Por ejemplo esto puede pasar si el fitness inicial es muy pequeño.

3. Enfriamiento muy rápido: Si el número de evaluaciones máximas maxevals es bajo (como 10000), el algoritmo se enfría en pocas iteraciones y ya no acepta malas soluciones.

**5. ¿Cual es mejor, ILS o ILS-ES?**

Como ya hemos comentado, la diferencia entre ILS y ILS-ES, es que en el segundo en vez de aplicar la BLRandom a las soluciones mutadas, aplica el algoritmo de enfiramiento simulado, ya hemos discutido que tanto Blrandom como ES obtienen resultados similares, por lo que tiene sentido que tanto ILS como ILS-ES obtengan resultadossimilares también, aunque un poco mejores en ILS, lo que nos puede indicar que la BL es más potente que hacer solo un intercambio, cuando la BL no entra en óptimos locale, lo que se reduce en ILS, ya que vamos aplicando mutación a las soluciones durante 10 iteraciones.

**6. ¿Qué algoritmos ofrecen mejor rendimiento, los de esta práctica, o los de la práctica 2?**

Para responder a esta pregunta, hemos tomado el mejor algoritmo de la práctica anterior(AM-(10,1) tenia una menor desviación media pero tarda mucho más), podemos ver que mejora en cuanto a desviación media a todos ls algoritmos de esta práctica,aunque a los mejores no por mucho, pero en cuanto a tiempo los de esta práctica son mucho más rápidos.

Estos resultados tienen sentido ya que el AM genera una diversidad de soluiciones much mayor que cualquiea de los algoritmos de esta práctica y además usa la búsqueda local, y evidentemente, esto provoca también que su tiempo de ejecución sea mayor.

Podemos decir, que en relación calidad tiempo, sería más sensato quedarnos con ILS antes que con AM-(10,0.1), ya que, a pesar de obtener una desviación ligeramente mayor, tarda muchisimo menos en ejecutarse.

**Conclusión Final**

A partir de los resultados obtenidos, podemos concluir que los algoritmos basados en *técnicas de búsqueda por trayectorias* logran una mejora significativa respecto a los enfoques más básicos como la búsqueda local aleatoria (**BLrandom**) o las heurísticas constructivas simples como **Greedy**. En particular, métodos como **BMB**, **ILS** y **GRASP-SIBL** consiguen combinar de forma efectiva la exploración del espacio de soluciones con una explotación más intensiva, evitando quedarse atrapados en óptimos locales de forma prematura gracias a estrategias como la re-inicialización, la mutación controlada o el uso de heurísticas de construcción.

Dentro de estos, el algoritmo **ILS** destaca especialmente por ofrecer un equilibrio muy sólido entre la calidad de la solución y el tiempo de ejecución. Frente a otros métodos más costosos como el **Algoritmo Memético AM-(10,0.1)**, ILS logra desviaciones medias solo ligeramente superiores, pero con un coste computacional muy inferior. Esto lo convierte en una opción preferible en contextos donde el tiempo de ejecución es un factor limitante.

Por otro lado, el algoritmo de **Enfriamiento Simulado (ES)**, a pesar de ser con-

ceptualmente potente, no ha mostrado ventajas claras sobre **BLrandom** en estos experimentos. Esto puede deberse a varias razones, entre ellas el uso de un vecindario demasiado limitado (solo se cambia un elemento por iteración), una diferencia insuficiente entre la temperatura inicial y final, o un enfriamiento demasiado rápido. Estos factores limitan su capacidad para escapar de óptimos locales, lo que enfatiza la importancia de una buena implementación y ajuste de parámetros en algoritmos estocásticos.

Finalmente, al comparar con los algoritmos de la práctica anterior, como los **Algoritmos Meméticos**, se observa que aunque estos pueden alcanzar mejores resultados en términos de calidad absoluta, lo hacen a costa de tiempos de ejecución significativamente mayores. Por tanto, desde un punto de vista de *eficiencia global* (relación calidad-tiempo), los algoritmos desarrollados en esta práctica —y en particular **ILS**— ofrecen un rendimiento altamente competitivo, siendo una alternativa muy válida frente a métodos más sofisticados y costosos.