UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 2. 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

1 de mayo de 2025

Índice

1. Descripción del problema **2**

2. Consideraciones comunes de los algoritmos empleados **2**
2.1. Clases solution y util . . . 3
2.2. Clase MH . . . 3
2.3. Operadores comunes a los algoritmos . . . 3
2.3.1. Seleccion . . . 4
2.3.2. Cruce . . . 5
2.3.3. Mutación . . . 8
2.4. Clase mddp . . . 9
2.4.1. La factorización . . . 11

3. Pseudocódigo de cada algoritmo **13**
3.1. Algoritmo AGG . . . 13
3.2. AGE . . . 15
3.3. AM . . . 18

4. Estructura del código de la práctica **20**
4.1. Compilar y ejecutar . . . 20

5. Experimentos y análisis de resultados **21**
5.1. Casos del problema empleados . . . 21
5.2. Resultados obtenidos y análisis de los mismos . . . 21

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 tres algoritmos: Algoritmo Genético Generacional(AGG), Algoritmo Genético Estacionario(AGE) y Algoritmo Memético(AM) que comentaremos en la siguiente sección. Estos tres algoritmos 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 AGG como AGE heredan de la clase MH y AM hereda de AGG, 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.

2.3. Operadores comunes a los algoritmos

En esta sección vamos a hablar de los operadores de selección, cruce y mutacioń de los tres algoritmos. Estos tres operadores son similares en el caso del AGG y AGE

con pequeñas variaciones en la implementación como veremos y AM al heredar de AGG usa los métodos de la clase padre.

2.3.1. Seleccion

En el caso del AGG, la selección consiste en un torneo en el que se escogen tres soluciones de manera aleatoria(se puede repetir) y te quedas con la mejor(la que tiene un menor fitness), y esto se repite 50 veces, luego te sigues quedando con 50 soluciones que pasan a la fase de cruce. Es importante recalcar que para el cruce las soluciones deben de tener un formato binario(en cada posición de la solución un valor de 1 indica que se escoge ese nodo y un valor de 0 indica que no se escoge), al final se hace un cambio de formato de las soluciones.

Pseudocódigo función torneo (AGG)

```
Funcion torneo(&soluciones_sel, &fitness_sel, size_poblacion,
size, soluciones_inicial, fitness_inicial, m, n_cruces)
  Para i desde 0 hasta size_poblacion - 1 Hacer:
    index1 ← valor aleatorio entre 0 y size_poblacion - 1
    index2 ← valor aleatorio entre 0 y size_poblacion - 1
    index3 ← valor aleatorio entre 0 y size_poblacion - 1
    best_index ← index1
    best_fitness ← fitness_inicial[index1]
    Si fitness_inicial[index2] <best_fitness Entonces:
      best_index ← index2
      best_fitness ← fitness_inicial[index2]
    FinSi
    Si fitness_inicial[index3] <best_fitness Entonces:
      best_index ← index3
    FinSi
    new_sol ← vector de tamaño size con ceros
    Para j desde 0 hasta m - 1 Hacer:
      posi ← soluciones_inicial[best_index][j]
      new_sol[posi] ← 1
    FinPara
    Añadir new_sol a soluciones_sel
    Añadir fitness_inicial[best_index] a fitness_sel
    Si i <2 * n_cruces Entonces:
      fitness_sel[i] ← -1
    FinSi
  FinPara
FinFuncion
```

Para el caso del AGE, la diferencia radica en que solo escogemos dos soluciones de la generación anterior, las cruzamos, se mutan con cierta probabilidad y en la fase de reemplazo en el caso de que el mejor de los padres sea mejor que la peor de la generación anterior la sustituye, si no no sucede nada, por ello no es necsario almacenar el fitness de los dos padres que se escogen en la selección y si es neceario

almacenar la peor solución de cada generación(esto último se actualiza en la fase de reemplazo que veremos en la sección del AGE).

2.3.2. Cruce

Tanto para el AGG y el AGE no se cruzan todas las soluciones. Vamos a distinguir dos tipos de cruces.

1. Cruce uniforme con reparación: Se generan dos hijos a partir de dos padres. Los valores comunes son mantenidos en ambos hijos y el resto de posiciones se dividen aleatoriamente para una solución hija o la otra. Es posible que al terminar el número de nodos puede no ser correcto, será necesario añadir o eliminar hasta tener el tamaño correcto de la solución. Para ello se implementa un método reparación que ahora comentaremos.
2. Cruce de intercambio o posición: Se generan dos hijos a partir de dos padres. Los valores comunes son mantenidos en ambos hijos y el resto de unos se reparte aleatoriamente en el resto de posiciones de tal manera que al final cada uno de los dos hijos tiene el número correcto de unos y no es necesario realizar reparación.

En el caso del AGG tenemos de la selección una población de 50 cromosomas (25 parejas) y una probabilidad de cruce de 0.7, luego cruzarán $0,7 * 25 = 17{,}5$ parejas $\Rightarrow$ 18 parejas(redondemos hacia arriba).

Es importante decir también que se cruza la primera solucioń con la segunda, la tercera con la cuarta, y así sucesivamente, hasta llegar a 18 cruces, luego se cruzan 36 cromosomas.

Para el AGE el procedimiento es análogo pero en vez de que se realizen 18 cruces se realiza solo 1(entre los dos padres), se comenta esto para no tener que incluir también el pseudocódigo del cruce del AGE.

Pseudocódigo función cruceUniforme

```
Funcion cruceUniforme(n_cruces, soluciones_sel, realproblem, m)
  info ← vector vacío
  contador_unos ← vector de tamaño 2*n_cruces con ceros
  size ← realproblem.obtenerTamañoProblema()
  Para i desde 0 hasta 2*n_cruces - 1 con paso 2 Hacer:
    Para j desde 0 hasta size - 1 Hacer:
      Si soluciones_sel[i][j] ≠ soluciones_sel[i+1][j] Entonces:
        c ← valor aleatorio entre 0 y 1
        soluciones_sel[i][j] ← c
        Si c = 0 Entonces:
          c ← c + 1
          contador_unos[i+1] ← contador_unos[i+1] + 1
        Sino:
          c ← c - 1
          contador_unos[i] ← contador_unos[i] + 1
```

```
        FinSi
        soluciones_sel[i+1][j] ← c
      Sino si soluciones_sel[i][j] = 1 Entonces:
        contador_unos[i] ← contador_unos[i] + 1
        contador_unos[i+1] ← contador_unos[i+1] + 1
      FinSi
    FinPara
    Añadir
realproblem.generarInfoFactoringBinario(soluciones_sel[i]) a
info
    Añadir
realproblem.generarInfoFactoringBinario(soluciones_sel[i+1]) a
info
  FinPara
  Llamar a reparacion(n_cruces, contador_unos, m, info,
soluciones_sel, realproblem)
FinFuncion
```

Ahora vamos a comentar en que consiste la reparación, la tenemos que hacer para las soluciones que se han cruzado, en el caso de que acaben el cruce con un número de unos menor que m, realizamos la siguiente **heurística**:

En el cruce, para cada solucion que se ha cruzado se ha almacenado su información de factorización(la factorización la comentaremos en la siguiente sección). Lo que hacemos en reparación es calcular la media de las distancias de cada nodo de la solución al resto(lo que se almacena en info) y añadimos un uno en la posición que previamente haya un 0 y que al añadir un 1 genere una menor dispersión, i.e. su suma de distancias respecto a la media sea la menor. Repetimos el procedimiento hasta que el número de unos llegue a m.

En el caso de que el número de unos sea mayor que m, la **heurística** es similar pero quitamos el nodo que genere una mayor dispersión respecto a la media. Repetimos el procedimiento hasta que el número de unos llegue a m.

Pseudocódigo función reparacion

```
Funcion reparacion(n_cruces, contador_unos, m, info,
&soluciones_sel, realproblem)
  size ← realproblem.obtenerTamañoProblema()
  Para i desde 0 hasta 2*n_cruces - 1 Hacer:
    Si contador_unos[i] <m Entonces:
      Mientras contador_unos[i] <m Hacer:
        media ← info[i].media()
        candidato ← indefinido
        dispersion_candidato ← -1
        Para j desde 0 hasta size - 1 Hacer:
          Si soluciones_sel[i][j] = 0 Entonces:
            disp ← |media -
```

```
realproblem.distanciasIn(soluciones_sel[i], j)|
            Si dispersion_candidato = -1 o disp
<dispersion_candidato Entonces:
              candidato ← j
              dispersion_candidato ← disp
            FinSi
          FinSi
        FinPara
        contador_unos[i] ← contador_unos[i] + 1
        realproblem.actualizarInfoFactoringIn(info[i],
soluciones_sel[i], candidato)
        soluciones_sel[i][candidato] ← 1
      FinMientras
    Sino si contador_unos[i] >m Entonces:
      Mientras contador_unos[i] >m Hacer:
        media ← info[i].media()
        candidato ← indefinido
        dispersion_candidato ← -1
        Para j desde 0 hasta size - 1 Hacer:
          disp ← |media - info[i].info[j]|
          Si dispersion_candidato = -1 y soluciones_sel[i][j] =
1 Entonces:
            candidato ← j
            dispersion_candidato ← disp
          Sino si soluciones_sel[i][j] = 1 y disp
>dispersion_candidato Entonces:
            candidato ← j
            dispersion_candidato ← disp
          FinSi
        FinPara
        contador_unos[i] ← contador_unos[i] - 1
        realproblem.actualizarInfoFactoringOut(info[i],
soluciones_sel[i], candidato)
        soluciones_sel[i][candidato] ← 0
      FinMientras
    FinSi
  FinPara
FinFuncion
```

Pseudocódigo función crucePosicion

```
Funcion crucePosicion(n_cruces, &soluciones_sel, size)
  Para i desde 0 hasta 2*n_cruces - 1 con paso 2 Hacer:
    Definir aux1 como un vector vacío
    Definir aux2 como un vector vacío
    Para j desde 0 hasta size - 1 Hacer:
```

```
      Si soluciones_sel[i][j] ≠ soluciones_sel[i+1][j] Entonces:
        Agregar soluciones_sel[i][j] a aux1
        soluciones_sel[i][j] ← -1
        Agregar soluciones_sel[i+1][j] a aux2
      FinSi
    FinPara
    Mezclar aleatoriamente aux1
    Mezclar aleatoriamente aux2
    Para j desde 0 hasta size - 1 Hacer:
      Si soluciones_sel[i][j] = -1 Entonces:
        soluciones_sel[i][j] ← ÚltimoElemento(aux1)
        Eliminar ÚltimoElemento de aux1
        soluciones_sel[i+1][j] ← ÚltimoElemento(aux2)
        Eliminar ÚltimoElemento de aux2
      FinSi
    FinPara
  FinPara
FinFuncion
```

2.3.3. Mutación

Para la etapa de mutación, en el caso de AGG cada cromosoma muta con una probabilidad del 10 %, luego cogemos 5 numeros aleatorio de las 50 soluciones, puede haber repeticion, y de cada uno de esos se coge un elemento de manera aletaoria y se cambia. Cuando ponemos el fitness de una solucón a -1 es para indicar qe luego hay que actualizarlo, y asi no hay que comprobar el fitness de todos los cromosomas.

Pseudocódigo función mutación (AGG)

```
Funcion mutacion(soluciones_sel, size, fitness_sel,
size_poblacion)
  Para i desde 0 hasta 4 Hacer:
    index_mut ← valor aleatorio entre 0 y size_poblacion - 1
    indices_ceros ← lista vacía
    indices_unos ← lista vacía
    Para j desde 0 hasta size - 1 Hacer:
      Si soluciones_sel[index_mut][j] = 0 Entonces:
        Añadir j a indices_ceros
      Sino:
        Añadir j a indices_unos
      FinSi
    FinPara
    idx_cero ← valor aleatorio entre 0 y tamaño(indices_ceros) -
1
    idx_uno ← valor aleatorio entre 0 y tamaño(indices_unos) - 1
    Intercambiar
```

```
soluciones_sel[index_mut][indices_ceros[idx_cero]] y
soluciones_sel[index_mut][indices_unos[idx_uno]]
    fitness_sel[index_mut] ← -1
  FinPara
FinFuncion
```

Para el caso del AGE, cada no de los dos padres tiene una probablidad del 10 % de mutar.

Pseudocódigo función mutación (AGE)

```
Funcion mutacion(padre, size)
  mut ← valor aleatorio entre 1 y 100
  Si mut ≤ 10 Entonces:
    indices_ceros ← lista vacía
    indices_unos ← lista vacía
    Para j desde 0 hasta size - 1 Hacer:
      Si padre[j] = 0 Entonces:
        Añadir j a indices_ceros
      Sino:
        Añadir j a indices_unos
      FinSi
    FinPara
    idx_cero ← valor aleatorio entre 0 y tamaño(indices_ceros) -
1
    idx_uno ← valor aleatorio entre 0 y tamaño(indices_unos) - 1
    Intercambiar padre[indices_ceros[idx_cero]] y
padre[indices_unos[idx_uno]]
  FinSi
FinFuncion
```

2.4. 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
```

2.4.1. La factorización

En la práctica anterior, ya comentamos que para la Búsqueda Local, cada vez que exploremos un vecindario, tenemos que calcular la dispersión de la nueva posible solución para comprobar si es menor. Si tenemos que llamar al método fitness que acabamos de desarrollar, se puede volver un algoritmo muy costoso, para mejorarlo, vamos a implementar la factorización, que se basa en que **no es necesario recalcular todas las distancias de la función objetivo:**

1. Al añadir un elemento, las distancias entre los que ya estaban en la solución se mantienen y basta con calcular la dispersión por el nuevo elemento al resto de elementos seleccionados
2. Al eliminar un elemento, las distancias entre elementos que se quedan en la solución se mantienen y basta con restar la distancia del elemento eliminado al resto de elementos en la solución

Para esta práctica, la factorización la vamos a usar para implementar la heurística de la reparación en el cruce uniforme, como ya hemos comentado, en el método cruce uniforme, para cad solución que se ha cruzado, se crea su **info** con el método **generateFactoringInfoBinary** que devuelve un objeto de la clase MddpFactoringInfo que tieneun tamaño **size** y en las posiciones donde la solución tenga un 1, calcula la suma de distancias al resto de nodos.

Pseudocódigo función generateFactoringInfoBinary

```
Funcion generateFactoringInfoBinary(solution_binary) devuelve
info
  Definir info como un objeto de tipo MddpFactoringInfo
  Para i desde 0 hasta tamaño(solution_binary) - 1 Hacer:
    Si solution_binary[i] = 1 Entonces:
      delta ← 0.0
      Para j desde 0 hasta tamaño(solution_binary) - 1 Hacer:
        Si solution_binary[j] = 1 Entonces:
          delta ← delta + matrix[i][j]
        FinSi
      FinPara
      Añadir delta a info.info
    Sino:
      Añadir -1.0 a info.info
    FinSi
  FinPara
  return info
FinFuncion
```

Esta información se pasa al método reparción com un vector de MddpFactoringInfo y vamos a usarlos siguiente métodos:

Para calcular la media de las distancias usamos el método media de la clase MddpFactoringInfo

Pseudocódigo función media

```
Funcion media() devuelve media
  Definir media ← 0.0
  Definir size ← tamaño(info)
  Definir total ← 0
  Para i desde 0 hasta size - 1 Hacer:
    Si info[i] ≠ -1 Entonces:
      media ← media + info[i]
      total ← total + 1
    FinSi
  FinPara
  return (media / total)
FinFuncion
```

Para el caso en el uee número de 1 sea menor que m. Hay que comprobar como afectaria a la dispersion introducir un determinado elemnto en la solución, para ello el método distanciasIn nos da la suma de distancias del posible nuevo elemento.

Pseudocódigo función distanciasIn

```
Funcion distanciasIn(solution, pos_change) devuelve delta
  Definir delta ← 0.0
  Para i desde 0 hasta tamaño(solution) - 1 Hacer:
    Si solution[i] = 1 Entonces:
      delta ← delta + matrix[i][pos_change]
    FinSi
  FinPara
  return delta
FinFuncion
```

Una vz que, tanto en el caso de tener un menor número de 1 que m hemos enconrado que numero añadir como en el caso contrario hemos encontrado que numero quitar llamamos respectivamente a updateSolutionFactoringInfoIn y updateSolutionFactoringInfoOut que lo que hacen es actualizar los valores de **info**.

Pseudocódigo función updateSolutionFactoringInfoIn

```
Funcion updateSolutionFactoringInfoIn(info, solution,
pos_change)
  Para i desde 0 hasta tamaño(info.info) - 1 Hacer:
    Si i ≠ pos_change y info.info[i] ≠ -1 Entonces:
      info.info[i] ← info.info[i] + matrix[i][pos_change]
    Sino si i = pos_change Entonces:
      Definir delta ← 0.0
      Para j desde 0 hasta tamaño(solution) - 1 Hacer:
        Si j ≠ pos_change Entonces:
          delta ← delta + matrix[pos_change][j]
```

```
      FinSi
      FinPara
      info.info[i] ← delta
    FinSi
  FinPara
FinFuncion
```

Pseudocódigo función updateSolutionFactoringInfoOut

```
Funcion updateSolutionFactoringInfoOut(info, solution,
pos_change)
  Para i desde 0 hasta tamaño(info.info) - 1 Hacer:
    Si i ≠ pos_change y info.info[i] ≠ -1 Entonces:
      info.info[i] ← info.info[i] - matrix[i][pos_change]
    Sino si i = pos_change Entonces:
      info.info[i] ← -1.0
    FinSi
  FinPara
FinFuncion
```

3. Pseudocódigo de cada algoritmo

Como ya hemos indicado, los tres 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 tres algoritmos y de las operaciones relevantes de cada uno de ellos.

3.1. Algoritmo AGG

Importante: Tanto para el AGG(y en consecuenciapara el AM) y para el AGE no se ha podido seguir al pie de la letra la API debido al uso de los metodos comentados en la sección de factorización, y definidos en la clase Mddp que hereda de la clase Problem.

Para el algoritmo AGG, partimos de un generación inicial formada por 50 cromsomas, y en cada iteración se realiza el torneo, cruce y mutación explicados anteriormente, además del reemplazo al final de la generación para iniciar la siguiente. En el reemplazo, la poblacion de los hijos sustituye automaticamente a la actual, para conservar el elitismo, si la mejor solucion de la nueva solucion es peor que la mejor solucion dde la anterior poblacion, la mejor solucion de la anterior poblacion sustituye a la mejor solucion de la nueva poblacion.

Además, retomamos la configuracion inicial de las soluciones y actualizamos los fitness en el cas de que sea necesario.

También es importnte comentar qe en todo momento se almacena la mejor solución de la población, que se actualiza en el remplazo, para al final del AGG devolverla.

Pseudocódigo función reemplazo (AGG)

```
Funcion reemplazo(&soluciones_inicial, &fitness_inicial,
size_poblacion, soluciones_sel, fitness_sel, &mejor_fitness,
&mejor_indice, size, &evaluaciones, realproblem)
  aux ← soluciones_inicial[mejor_indice]
  nuevo_mejor_fitness ← -1
  nuevo_mejor_indice ← indefinido
  Para i desde 0 hasta size_poblacion - 1 hacer:
    new_sol ← vector vacío
    Para j desde 0 hasta size - 1 hacer:
      Si soluciones_sel[i][j] = 1 Entonces:
        Añadir j a new_sol
      FinSi
    FinPara
    soluciones_inicial[i] ← new_sol
    fitness_inicial[i] ← fitness_sel[i]
    Si fitness_inicial[i] = -1 Entonces:
      fitness_inicial[i] ←
realproblem.fitness(soluciones_inicial[i])
      evaluaciones ← evaluaciones + 1
    FinSi
    Si nuevo_mejor_fitness = -1 o fitness_inicial[i]
<nuevo_mejor_fitness Entonces:
      nuevo_mejor_fitness ← fitness_inicial[i]
      nuevo_mejor_indice ← i
    FinSi
  FinPara
  Si mejor_fitness <nuevo_mejor_fitness Entonces:
    soluciones_inicial[0] ← aux
  Sino:
    mejor_fitness ← nuevo_mejor_fitness
    mejor_indice ← nuevo_mejor_indice
  FinSi
FinFuncion
```

Con todo esto el método optimize queda así:

Pseudocódigo función optimize (AGG)

```
Funcion optimize(problem, maxevals)
  Asegurar que maxevals >0
  realproblem ← problem casteado como Mddp
  m ← tamaño de la solución (realproblem.getSolutionSize())
  size ← tamaño del problema
  evaluaciones ← 0
  soluciones_inicial ← vector vacío
```

```
  fitness_inicial ← vector vacío
  size_poblacion ← 50
  mejor_fitness ← -1
  mejor_indice ← indefinido
  Para i desde 0 hasta size_poblacion - 1 hacer:
    aux ← realproblem.createSolution()
    Añadir aux a soluciones_inicial
    fitness ← realproblem.fitness(aux)
    Añadir fitness a fitness_inicial
    Si mejor_fitness = -1 o fitness <mejor_fitness Entonces:
      mejor_fitness ← fitness
      mejor_indice ← i
    FinSi
  FinPara
  evaluaciones ← size_poblacion
  Mientras evaluaciones <maxevals hacer:
    n_cruces ← 18
    soluciones_sel ← vector vacío
    fitness_sel ← vector vacío
    torneo(soluciones_sel, fitness_sel, size_poblacion, size,
soluciones_inicial, fitness_inicial, m, n_cruces)
    Si cruce_uniforme Entonces:
      cruceUniforme(n_cruces, soluciones_sel, realproblem, m)
    Sino:
      crucePosicion(n_cruces, soluciones_sel, size)
    FinSi
    mutacion(soluciones_sel, size, fitness_sel, size_poblacion)
    reemplazo(soluciones_inicial, fitness_inicial,
size_poblacion, soluciones_sel, fitness_sel, mejor_fitness,
mejor_indice, size, evaluaciones, realproblem)
  FinMientras
  Return ResultMH(soluciones_inicial[mejor_indice],
fitness_inicial[mejor_indice], evaluaciones)
FinFuncion
```

3.2. AGE

Para el algoritmo AGE, partimos de un generación inicial formada por 50 cromsomas, y en cada iteración se realiza el torneo, cruce y mutación explicados anteriormente, además del reemplazo al final de la generación para iniciar la siguiente. En el reemplazo, los dos descendientes generados tras el cruce y la mutacion compiten entre si, y el mejor de ellos sustituye a la peor solucion de la poblacion actual, en caso de ser mejor que ella.

Además, retomamos la configuracion inicial de las soluciones y actualizamos los fitness en el cas de que sea necesario.

También es importnte comentar qe en todo momento se almacena la mejor solución

de la población, que se actualiza en el remplazo, para al final del AGG devolverla. Además, por como realizamos el método de reemplazo también almacenamos la peor solución de cda generación

Pseudocódigo función reemplazo (AGE)

```
Funcion reemplazo(padres, realproblem, size, evaluaciones,
mejor_fitness, mejor_indice, peor_fitness, index_peor,
fitness_inicial, soluciones_inicial)
  Para i desde 0 hasta 1 hacer:
    new_sol ← vector vacío
    Para j desde 0 hasta size - 1 hacer:
      Si padres[i][j] = 1 Entonces:
        Añadir j a new_sol
      FinSi
    FinPara
    padres[i] ← new_sol
  FinPara
  fitness1 ← realproblem.fitness(padres[0])
  fitness2 ← realproblem.fitness(padres[1])
  evaluaciones ← evaluaciones + 2
  index_reemplazamiento ← 0
  reemplazamiento ← fitness1
  Si fitness2 <reemplazamiento Entonces:
    reemplazamiento ← fitness2
    index_reemplazamiento ← 1
  FinSi
  Si reemplazamiento <mejor_fitness Entonces:
    mejor_fitness ← reemplazamiento
    mejor_indice ← index_reemplazamiento
  FinSi
  Si reemplazamiento <peor_fitness Entonces:
    soluciones_inicial[index_peor] ←
padres[index_reemplazamiento]
    fitness_inicial[index_peor] ← reemplazamiento
    // Actualizar peor fitness
    primer ← true
    Para i desde 0 hasta tamaño de fitness_inicial - 1 hacer:
      Si primer Entonces:
        index_peor ← i
        peor_fitness ← fitness_inicial[i]
        primer ← false
      Sino si fitness_inicial[i] >peor_fitness Entonces:
        index_peor ← i
        peor_fitness ← fitness_inicial[i]
      FinSi
    FinPara
```

```
  FinSi
FinFuncion
```

Con todo esto el método optimize queda así:

Pseudocódigo función optimize (AGE)

```
Funcion optimize(problem, maxevals)
  Asegurar que maxevals >0
  realproblem ← convertir problem a tipo Mddp
  m ← realproblem.getSolutionSize()
  size ← realproblem.getProblemSize()
  evaluaciones ← 0
  soluciones_inicial ← vector vacío
  fitness_inicial ← vector vacío
  size_poblacion ← 50
  mejor_fitness ← -1
  peor_fitness ← indefinido
  mejor_indice, index_peor ← indefinido
  Para i desde 0 hasta size_poblacion - 1 hacer:
    aux ← realproblem.createSolution()
    fitness ← realproblem.fitness(aux)
    Añadir aux a soluciones_inicial
    Añadir fitness a fitness_inicial
    Si mejor_fitness = -1 Entonces:
      mejor_fitness ← fitness
      mejor_indice ← i
      peor_fitness ← fitness
      index_peor ← i
    Sino si fitness <mejor_fitness Entonces:
      mejor_fitness ← fitness
      mejor_indice ← i
    Sino si fitness >peor_fitness Entonces:
      peor_fitness ← fitness
      index_peor ← i
    FinSi
  FinPara
  evaluaciones ← size_poblacion
  Mientras evaluaciones <maxevals hacer:
    padres ← vector vacío
    torneo(padres, size_poblacion, fitness_inicial,
soluciones_inicial, size, m)
    Si cruce_uniforme Entonces:
      cruceUniforme(padres, realproblem, m)
    Sino:
      crucePosicion(padres, size)
    FinSi
```

```
    mutacion(padres[0], size)
    mutacion(padres[1], size)
    reemplazo(padres, realproblem, size, evaluaciones,
mejor_fitness, mejor_indice, peor_fitness, index_peor,
fitness_inicial, soluciones_inicial)
  FinMientras
  retornar ResultMH(soluciones_inicial[mejor_indice],
mejor_fitness, evaluaciones)
FinFuncion
```

3.3. AM

Como ya comentamos el AM hereda del AGG. Dispone de un objeto privado de la clase BusquedaLocalPM para llamar a su método optimize, y al igual que las clases de los algoritmos AGG y AGE tenían un atributo para distinguir entre los dos tipos de cruces, aqui disponemos de un entero, tipo_hibridacion para distinguir a que conjunto de la población se aplica la búsqueda local aleotoria. tipo_hibridacion puede valer 1,2 o 3, si vale 1: AM-(10,1.0), si vale 2: AM-(10,0.1) y si vale 3: AM-(10,0.1mej).

1. AM-(10,1.0): Cada 10 generaciones se aplica la BLrandom al 100 % de la población.
2. AM-(10,0.1): Cada 10 generaciones se aplica la BLrandom al 10 % de la población de manera aleatoria.
3. AM-(10,0.1mej): Cada 10 generaciones se aplica la BLrandom al 100 % mejor de la población.

Recordemos que el algoritmo memético implmenta el algoritmo generacional y cada cierto número de generaciones, en este caso 10, llama a la buśqueda local para comprobar si modificando un poco la solución esta mejora.

Otro detalle importante es que para el caso AM-(10,0.1mej), necesitamos almacenar generación a generación los cinco mejores cromosomas, para ello hemos usado una priority_queue¡ParSolucion¿donde ParSolucion es un strcut que almacena una solución, su fitness y el indice que tiene en la poblacion, además de un operador de comparación, de tal manera que en lo alto de la cola este la peor solución de las 5, y si encontramos una solución mejor, la sustituyamos en reemplazo:

```
struct ParSolucion {
    tSolution solucion;
    tFitness fitness;
    int index;

    bool operator<(const ParSolucion &other) const {
        return fitness < other.fitness;
    }
};
```

Para comprobar si hay una solución que se pueda agregar a las 5 mejores disponemos de la función AgregaraMejores, cuyo psudocódigo es el siguiente:

Pseudocódigo función AgregarAMejores (AM)

```
Funcion AgregarAMejores(&mejores_5, sol, fit, index)
  Si tamaño de mejores_5 <5 Entonces:
    Insertar (sol, fit, index) en mejores_5
  Sino si fit <mejores_5.tope().fitness Entonces:
    Eliminar el elemento de mayor fitness (tope) de mejores_5
    Insertar (sol, fit, index) en mejores_5
  FinSi
FinFuncion
```

Como ya hemos comentado, el método de reemplazo es prácticamente idéntico que el del padre AGG, pero para mantener la información de las cinco mejores soluciones hemos sobreescrito reemplazo y hemos añadido la siguiente liñea:

```
AgregarAMejores(mejores_5, new_sol, fitness_inicial[i], i);
```

Donde mejores_5 es la cola con la información de las cinco mejores soluciones, new_sol la solución que queremos comprobar si es mejor que la peor que hay en mejores_5 y fitness_inicial[i] e i son el fitness y el índice de la población respectivamente de new_sol.

Con esto el método optimize queda así:

Pseudocódigo función optimize (AM)

```
Función optimize(problem, maxevals)
  Inicializar parámetros y estructuras auxiliares
  Inicializar población de tamaño 50: soluciones_inicial,
fitness_inicial
  Evaluar cada solución inicial, guardar mejores 5 con
AgregarAMejores
  Actualizar mejor_fitness y mejor_indice
  evaluaciones = 50, n_generaciones = 0
  Mientras evaluaciones <maxevals hacer:
    Si n_generaciones == 10:
      Resetear n_generaciones
      Según tipo_hibridacion hacer:
        Caso 1: aplicar BL a toda la población
        Caso 2: aplicar BL a 5 individuos aleatorios
        Caso 3: aplicar BL a los 5 mejores guardados
    Seleccionar 36 soluciones mediante torneo (para 18 cruces)
    Realizar cruce uniforme en las parejas
    Mutar 5 soluciones aleatorias (10% de la población)
    Reemplazar población con elitismo y actualizar mejores
    Incrementar n_generaciones
```

```
  FinMientras
  Devolver ResultMH(mejor solución, mejor fitness, evaluaciones)
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.

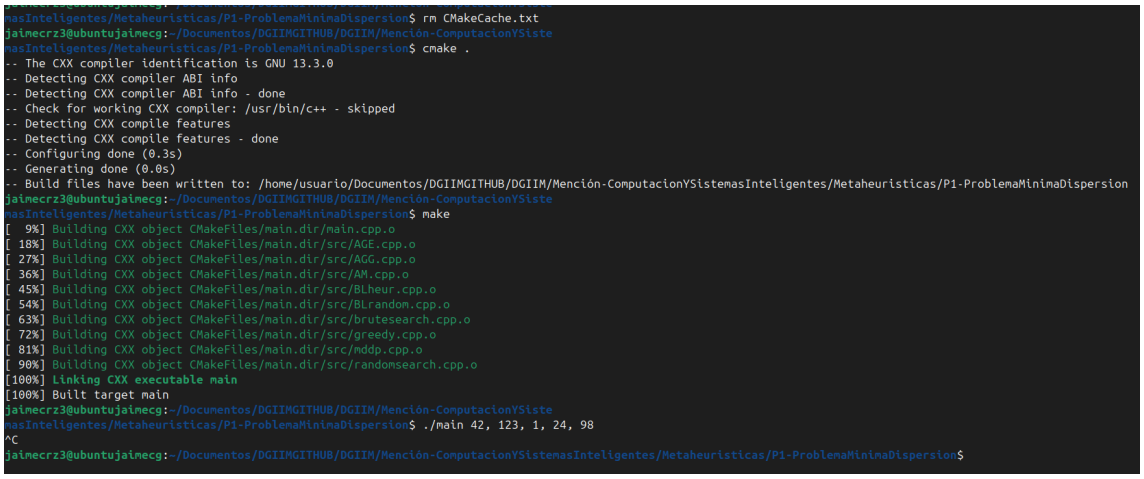

Figura 1: Manera de compilar y ejecutar la práctica

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 resultados para cada uno de los algoritmos(AGG con los dos cruces, AGE con los dos cruces, AM con los tres tipos de hibridacion) donde se recogen las desviaciones de los resultados de dicho algoritmo con respecto a la solución óptima y el tiempo en milisegundos:

Cuadro 1: Resultados del algoritmo AGG-uniforme

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0,00	1,13E+02
GKD-b_2_n25_m2	0,00	1,17E+02
GKD-b_3_n25_m2	0,00	1,16E+02
GKD-b_4_n25_m2	0,00	8,50E+01
GKD-b_5_n25_m2	0,00	8,40E+01

Caso	Desv	Tiempo
GKD-b_6_n25_m7	71,34	1,08E+02
GKD-b_7_n25_m7	66,84	1,07E+02
GKD-b_8_n25_m7	49,88	1,05E+02
GKD-b_9_n25_m7	44,67	1,07E+02
GKD-b_10_n25_m7	28,60	1,09E+02
GKD-b_11_n50_m5	84,88	1,25E+02
GKD-b_12_n50_m5	87,57	1,23E+02
GKD-b_13_n50_m5	88,52	1,25E+02
GKD-b_14_n50_m5	91,39	1,27E+02
GKD-b_15_n50_m5	82,88	1,22E+02
GKD-b_16_n50_m15	51,23	2,04E+02
GKD-b_17_n50_m15	44,25	2,06E+02
GKD-b_18_n50_m15	49,43	2,01E+02
GKD-b_19_n50_m15	41,74	2,04E+02
GKD-b_20_n50_m15	48,64	2,06E+02
GKD-b_21_n100_m10	76,40	2,33E+02
GKD-b_22_n100_m10	64,53	2,48E+02
GKD-b_23_n100_m10	66,03	2,38E+02
GKD-b_24_n100_m10	82,73	2,37E+02
GKD-b_25_n100_m10	61,76	2,36E+02
GKD-b_26_n100_m30	32,10	5,49E+02
GKD-b_27_n100_m30	29,24	6,92E+02
GKD-b_28_n100_m30	45,71	6,56E+02
GKD-b_29_n100_m30	56,61	5,68E+02
GKD-b_30_n100_m30	32,94	5,69E+02
GKD-b_31_n125_m12	80,35	3,13E+02
GKD-b_32_n125_m12	67,50	3,11E+02
GKD-b_33_n125_m12	67,47	3,04E+02
GKD-b_34_n125_m12	56,92	3,08E+02
GKD-b_35_n125_m12	66,58	3,10E+02
GKD-b_36_n125_m37	33,59	8,16E+02
GKD-b_37_n125_m37	44,03	8,08E+02
GKD-b_38_n125_m37	33,83	8,07E+02
GKD-b_39_n125_m37	34,35	8,14E+02
GKD-b_40_n125_m37	39,97	8,03E+02
GKD-b_41_n150_m15	67,87	4,15E+02
GKD-b_42_n150_m15	62,93	4,12E+02
GKD-b_43_n150_m15	63,85	4,10E+02
GKD-b_44_n150_m15	67,96	4,06E+02
GKD-b_45_n150_m15	67,76	4,13E+02
GKD-b_46_n150_m45	32,94	1,15E+03
GKD-b_47_n150_m45	31,59	1,14E+03
GKD-b_48_n150_m45	32,17	1,13E+03
GKD-b_49_n150_m45	39,59	1,13E+03
GKD-b_50_n150_m45	34,17	1,12E+03

Cuadro 2: Resultados del algoritmo AGG-posición

Caso	**Desv**	**Tiempo**
GKD-b_1_n25_m2	0,00	4,90E+01
GKD-b_2_n25_m2	0,00	4,90E+01
GKD-b_3_n25_m2	0,00	4,90E+01
GKD-b_4_n25_m2	0,00	4,80E+01
GKD-b_5_n25_m2	0,00	4,80E+01
GKD-b_6_n25_m7	64,04	5,50E+01
GKD-b_7_n25_m7	56,92	5,50E+01
GKD-b_8_n25_m7	59,30	5,40E+01
GKD-b_9_n25_m7	50,11	5,60E+01
GKD-b_10_n25_m7	32,30	5,50E+01
GKD-b_11_n50_m5	89,05	5,90E+01
GKD-b_12_n50_m5	87,82	5,90E+01
GKD-b_13_n50_m5	89,46	7,00E+01
GKD-b_14_n50_m5	90,80	8,60E+01
GKD-b_15_n50_m5	85,47	8,90E+01
GKD-b_16_n50_m15	70,79	1,20E+02
GKD-b_17_n50_m15	62,38	1,00E+02
GKD-b_18_n50_m15	64,15	7,90E+01
GKD-b_19_n50_m15	64,29	9,20E+01
GKD-b_20_n50_m15	59,63	1,13E+02
GKD-b_21_n100_m10	74,13	8,20E+01
GKD-b_22_n100_m10	74,89	8,20E+01
GKD-b_23_n100_m10	64,82	8,10E+01
GKD-b_24_n100_m10	82,64	8,00E+01
GKD-b_25_n100_m10	64,15	8,10E+01
GKD-b_26_n100_m30	59,81	1,37E+02
GKD-b_27_n100_m30	62,70	1,33E+02
GKD-b_28_n100_m30	72,85	1,38E+02
GKD-b_29_n100_m30	63,00	1,40E+02
GKD-b_30_n100_m30	62,73	1,35E+02
GKD-b_31_n125_m12	87,54	8,90E+01
GKD-b_32_n125_m12	71,73	8,90E+01
GKD-b_33_n125_m12	74,36	9,00E+01
GKD-b_34_n125_m12	73,49	9,00E+01
GKD-b_35_n125_m12	77,08	9,00E+01
GKD-b_36_n125_m37	60,74	1,80E+02
GKD-b_37_n125_m37	61,76	1,79E+02
GKD-b_38_n125_m37	67,99	1,77E+02
GKD-b_39_n125_m37	64,61	1,77E+02
GKD-b_40_n125_m37	64,00	1,76E+02
GKD-b_41_n150_m15	72,92	1,02E+02
GKD-b_42_n150_m15	73,38	1,01E+02
GKD-b_43_n150_m15	70,77	1,02E+02
GKD-b_44_n150_m15	66,56	1,02E+02

Caso	Desv	Tiempo
GKD-b_45_n150_m15	64,16	1,38E+02
GKD-b_46_n150_m45	68,30	2,62E+02
GKD-b_47_n150_m45	60,11	2,32E+02
GKD-b_48_n150_m45	62,49	2,27E+02
GKD-b_49_n150_m45	64,86	2,35E+02
GKD-b_50_n150_m45	65,34	2,31E+02

Cuadro 3: Resultados del algoritmo AGE-uniforme

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0,00	2,42E+02
GKD-b_2_n25_m2	0,00	2,45E+02
GKD-b_3_n25_m2	0,00	2,43E+02
GKD-b_4_n25_m2	0,00	2,42E+02
GKD-b_5_n25_m2	0,00	2,49E+02
GKD-b_6_n25_m7	58,22	2,05E+02
GKD-b_7_n25_m7	50,99	2,02E+02
GKD-b_8_n25_m7	52,71	1,97E+02
GKD-b_9_n25_m7	40,19	1,95E+02
GKD-b_10_n25_m7	35,69	1,96E+02
GKD-b_11_n50_m5	84,20	2,07E+02
GKD-b_12_n50_m5	85,08	2,07E+02
GKD-b_13_n50_m5	85,15	2,06E+02
GKD-b_14_n50_m5	90,54	2,26E+02
GKD-b_15_n50_m5	75,70	2,10E+02
GKD-b_16_n50_m15	56,26	2,83E+02
GKD-b_17_n50_m15	42,82	2,85E+02
GKD-b_18_n50_m15	53,81	2,93E+02
GKD-b_19_n50_m15	47,64	2,88E+02
GKD-b_20_n50_m15	48,40	2,91E+02
GKD-b_21_n100_m10	67,87	3,25E+02
GKD-b_22_n100_m10	69,83	3,25E+02
GKD-b_23_n100_m10	63,78	3,24E+02
GKD-b_24_n100_m10	82,36	3,26E+02
GKD-b_25_n100_m10	62,87	3,22E+02
GKD-b_26_n100_m30	36,67	6,53E+02
GKD-b_27_n100_m30	30,55	8,18E+02
GKD-b_28_n100_m30	47,85	6,45E+02
GKD-b_29_n100_m30	40,67	6,90E+02
GKD-b_30_n100_m30	41,30	6,69E+02
GKD-b_31_n125_m12	73,64	3,95E+02
GKD-b_32_n125_m12	69,28	4,01E+02
GKD-b_33_n125_m12	64,39	4,02E+02
GKD-b_34_n125_m12	65,51	3,96E+02
GKD-b_35_n125_m12	63,84	3,91E+02
GKD-b_36_n125_m37	37,76	8,79E+02

Caso	Desv	Tiempo
GKD-b_37_n125_m37	34,41	8,97E+02
GKD-b_38_n125_m37	39,36	8,79E+02
GKD-b_39_n125_m37	36,53	8,77E+02
GKD-b_40_n125_m37	43,28	8,80E+02
GKD-b_41_n150_m15	68,55	5,30E+02
GKD-b_42_n150_m15	65,45	5,03E+02
GKD-b_43_n150_m15	60,22	5,06E+02
GKD-b_44_n150_m15	60,30	4,97E+02
GKD-b_45_n150_m15	62,41	5,06E+02
GKD-b_46_n150_m45	40,04	1,23E+03
GKD-b_47_n150_m45	30,09	1,25E+03
GKD-b_48_n150_m45	39,52	1,23E+03
GKD-b_49_n150_m45	42,52	1,23E+03
GKD-b_50_n150_m45	24,55	1,21E+03

Cuadro 4: Resultados del algoritmo AGE-posicion

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0,00	1,56E+02
GKD-b_2_n25_m2	0,00	1,56E+02
GKD-b_3_n25_m2	0,00	1,58E+02
GKD-b_4_n25_m2	0,00	1,64E+02
GKD-b_5_n25_m2	0,00	1,60E+02
GKD-b_6_n25_m7	59,45	1,46E+02
GKD-b_7_n25_m7	52,67	1,45E+02
GKD-b_8_n25_m7	52,34	1,47E+02
GKD-b_9_n25_m7	53,64	1,47E+02
GKD-b_10_n25_m7	49,19	1,45E+02
GKD-b_11_n50_m5	86,53	1,50E+02
GKD-b_12_n50_m5	84,60	1,50E+02
GKD-b_13_n50_m5	85,62	1,50E+02
GKD-b_14_n50_m5	91,63	1,49E+02
GKD-b_15_n50_m5	78,09	1,51E+02
GKD-b_16_n50_m15	62,46	1,60E+02
GKD-b_17_n50_m15	61,35	1,68E+02
GKD-b_18_n50_m15	59,38	1,67E+02
GKD-b_19_n50_m15	62,27	1,65E+02
GKD-b_20_n50_m15	54,00	1,71E+02
GKD-b_21_n100_m10	72,76	1,68E+02
GKD-b_22_n100_m10	70,66	1,68E+02
GKD-b_23_n100_m10	70,38	1,71E+02
GKD-b_24_n100_m10	81,22	1,70E+02
GKD-b_25_n100_m10	69,37	1,67E+02
GKD-b_26_n100_m30	59,75	2,37E+02
GKD-b_27_n100_m30	66,93	2,29E+02
GKD-b_28_n100_m30	67,19	2,34E+02

Caso	Desv	Tiempo
GKD-b_29_n100_m30	62,09	2,31E+02
GKD-b_30_n100_m30	61,89	2,24E+02
GKD-b_31_n125_m12	80,31	1,79E+02
GKD-b_32_n125_m12	66,72	1,76E+02
GKD-b_33_n125_m12	74,84	1,78E+02
GKD-b_34_n125_m12	69,48	1,76E+02
GKD-b_35_n125_m12	78,82	1,78E+02
GKD-b_36_n125_m37	61,84	2,61E+02
GKD-b_37_n125_m37	61,06	2,62E+02
GKD-b_38_n125_m37	64,21	2,60E+02
GKD-b_39_n125_m37	61,55	2,55E+02
GKD-b_40_n125_m37	65,24	2,55E+02
GKD-b_41_n150_m15	67,41	1,91E+02
GKD-b_42_n150_m15	74,69	1,95E+02
GKD-b_43_n150_m15	74,03	1,92E+02
GKD-b_44_n150_m15	63,54	1,89E+02
GKD-b_45_n150_m15	74,74	1,88E+02
GKD-b_46_n150_m45	68,04	3,15E+02
GKD-b_47_n150_m45	63,42	3,19E+02
GKD-b_48_n150_m45	59,56	3,37E+02
GKD-b_49_n150_m45	66,74	3,57E+02
GKD-b_50_n150_m45	67,40	3,49E+02

Cuadro 5: Resultados del algoritmo AM-(10,1.0)

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0,00	1,14E+02
GKD-b_2_n25_m2	0,00	1,15E+02
GKD-b_3_n25_m2	0,00	1,17E+02
GKD-b_4_n25_m2	0,00	1,23E+02
GKD-b_5_n25_m2	0,00	1,22E+02
GKD-b_6_n25_m7	14,75	2,35E+02
GKD-b_7_n25_m7	55,08	2,49E+02
GKD-b_8_n25_m7	0,00	2,15E+02
GKD-b_9_n25_m7	0,00	1,93E+02
GKD-b_10_n25_m7	9,27	1,99E+02
GKD-b_11_n50_m5	68,27	2,58E+02
GKD-b_12_n50_m5	64,50	2,62E+02
GKD-b_13_n50_m5	42,63	2,63E+02
GKD-b_14_n50_m5	68,44	2,62E+02
GKD-b_15_n50_m5	39,68	2,62E+02
GKD-b_16_n50_m15	10,40	7,41E+02
GKD-b_17_n50_m15	0,00	7,49E+02
GKD-b_18_n50_m15	19,59	7,80E+02
GKD-b_19_n50_m15	2,27	9,17E+02
GKD-b_20_n50_m15	10,67	7,84E+02

Caso	Desv	Tiempo
GKD-b_21_n100_m10	30,56	1,18E+03
GKD-b_22_n100_m10	36,77	1,13E+03
GKD-b_23_n100_m10	30,79	1,19E+03
GKD-b_24_n100_m10	57,27	1,15E+03
GKD-b_25_n100_m10	21,47	1,12E+03
GKD-b_26_n100_m30	1,62	4,39E+03
GKD-b_27_n100_m30	4,67	4,96E+03
GKD-b_28_n100_m30	2,64	4,65E+03
GKD-b_29_n100_m30	12,57	4,59E+03
GKD-b_30_n100_m30	11,09	4,59E+03
GKD-b_31_n125_m12	39,93	1,66E+03
GKD-b_32_n125_m12	31,67	1,59E+03
GKD-b_33_n125_m12	29,54	1,57E+03
GKD-b_34_n125_m12	55,41	1,53E+03
GKD-b_35_n125_m12	22,16	1,69E+03
GKD-b_36_n125_m37	0,12	8,04E+03
GKD-b_37_n125_m37	15,46	8,67E+03
GKD-b_38_n125_m37	7,64	8,84E+03
GKD-b_39_n125_m37	11,47	8,35E+03
GKD-b_40_n125_m37	11,16	8,53E+03
GKD-b_41_n150_m15	34,98	2,68E+03
GKD-b_42_n150_m15	30,14	2,73E+03
GKD-b_43_n150_m15	21,96	2,65E+03
GKD-b_44_n150_m15	30,28	2,82E+03
GKD-b_45_n150_m15	24,16	2,88E+03
GKD-b_46_n150_m45	9,59	1,40E+04
GKD-b_47_n150_m45	6,58	1,39E+04
GKD-b_48_n150_m45	22,72	1,40E+04
GKD-b_49_n150_m45	9,12	1,42E+04
GKD-b_50_n150_m45	15,02	1,46E+04

Cuadro 6: Resultados del algoritmo AGM-(10,0.1)

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0,00	1,39E+02
GKD-b_2_n25_m2	0,00	1,49E+02
GKD-b_3_n25_m2	0,00	1,08E+02
GKD-b_4_n25_m2	0,00	1,17E+02
GKD-b_5_n25_m2	0,00	1,26E+02
GKD-b_6_n25_m7	11,75	1,60E+02
GKD-b_7_n25_m7	25,60	1,82E+02
GKD-b_8_n25_m7	19,25	1,78E+02
GKD-b_9_n25_m7	31,07	1,47E+02
GKD-b_10_n25_m7	14,83	1,76E+02
GKD-b_11_n50_m5	76,37	2,62E+02
GKD-b_12_n50_m5	77,90	2,66E+02

Caso	Desv	Tiempo
GKD-b_13_n50_m5	74,14	2,53E+02
GKD-b_14_n50_m5	81,21	2,72E+02
GKD-b_15_n50_m5	66,02	2,42E+02
GKD-b_16_n50_m15	28,11	4,20E+02
GKD-b_17_n50_m15	6,29	3,84E+02
GKD-b_18_n50_m15	30,92	3,63E+02
GKD-b_19_n50_m15	38,29	4,27E+02
GKD-b_20_n50_m15	28,74	4,02E+02
GKD-b_21_n100_m10	42,95	4,01E+02
GKD-b_22_n100_m10	53,08	5,27E+02
GKD-b_23_n100_m10	50,44	4,35E+02
GKD-b_24_n100_m10	87,05	3,79E+02
GKD-b_25_n100_m10	34,32	3,70E+02
GKD-b_26_n100_m30	13,07	1,28E+03
GKD-b_27_n100_m30	19,02	1,12E+03
GKD-b_28_n100_m30	17,39	1,20E+03
GKD-b_29_n100_m30	21,03	1,28E+03
GKD-b_30_n100_m30	17,77	1,39E+03
GKD-b_31_n125_m12	60,02	6,20E+02
GKD-b_32_n125_m12	48,34	5,93E+02
GKD-b_33_n125_m12	45,80	5,80E+02
GKD-b_34_n125_m12	42,34	5,66E+02
GKD-b_35_n125_m12	47,59	4,68E+02
GKD-b_36_n125_m37	16,20	2,04E+03
GKD-b_37_n125_m37	22,43	1,97E+03
GKD-b_38_n125_m37	23,90	1,93E+03
GKD-b_39_n125_m37	27,72	1,87E+03
GKD-b_40_n125_m37	24,23	1,89E+03
GKD-b_41_n150_m15	48,40	8,41E+02
GKD-b_42_n150_m15	43,08	7,82E+02
GKD-b_43_n150_m15	74,73	7,38E+02
GKD-b_44_n150_m15	37,17	7,19E+02
GKD-b_45_n150_m15	37,08	8,99E+02
GKD-b_46_n150_m45	28,39	3,02E+03
GKD-b_47_n150_m45	22,92	3,14E+03
GKD-b_48_n150_m45	18,74	2,90E+03
GKD-b_49_n150_m45	36,94	3,22E+03
GKD-b_50_n150_m45	16,21	2,96E+03

Cuadro 7: Resultados del algoritmo AM-(10,0.1mej)

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0,00	1,38E+02
GKD-b_2_n25_m2	0,00	1,34E+02
GKD-b_3_n25_m2	0,00	1,29E+02
GKD-b_4_n25_m2	0,00	1,55E+02

Caso	Desv	Tiempo
GKD-b_5_n25_m2	0,00	1,29E+02
GKD-b_6_n25_m7	56,16	1,50E+02
GKD-b_7_n25_m7	43,25	1,52E+02
GKD-b_8_n25_m7	37,21	2,01E+02
GKD-b_9_n25_m7	41,92	1,47E+02
GKD-b_10_n25_m7	22,12	1,70E+02
GKD-b_11_n50_m5	77,90	1,91E+02
GKD-b_12_n50_m5	82,86	2,02E+02
GKD-b_13_n50_m5	76,00	2,18E+02
GKD-b_14_n50_m5	82,04	2,13E+02
GKD-b_15_n50_m5	75,09	2,12E+02
GKD-b_16_n50_m15	42,60	2,89E+02
GKD-b_17_n50_m15	23,74	2,79E+02
GKD-b_18_n50_m15	35,90	2,72E+02
GKD-b_19_n50_m15	35,08	2,71E+02
GKD-b_20_n50_m15	25,63	2,71E+02
GKD-b_21_n100_m10	54,79	3,52E+02
GKD-b_22_n100_m10	56,62	3,96E+02
GKD-b_23_n100_m10	50,88	4,39E+02
GKD-b_24_n100_m10	74,89	4,34E+02
GKD-b_25_n100_m10	40,44	4,43E+02
GKD-b_26_n100_m30	11,91	1,10E+03
GKD-b_27_n100_m30	15,83	1,13E+03
GKD-b_28_n100_m30	33,80	1,08E+03
GKD-b_29_n100_m30	31,18	1,08E+03
GKD-b_30_n100_m30	25,84	1,14E+03
GKD-b_31_n125_m12	54,46	5,45E+02
GKD-b_32_n125_m12	48,33	5,42E+02
GKD-b_33_n125_m12	51,38	4,98E+02
GKD-b_34_n125_m12	50,08	5,84E+02
GKD-b_35_n125_m12	56,98	5,70E+02
GKD-b_36_n125_m37	27,90	1,77E+03
GKD-b_37_n125_m37	32,16	1,85E+03
GKD-b_38_n125_m37	25,86	1,68E+03
GKD-b_39_n125_m37	28,28	1,90E+03
GKD-b_40_n125_m37	35,20	1,95E+03
GKD-b_41_n150_m15	56,65	8,96E+02
GKD-b_42_n150_m15	41,92	8,09E+02
GKD-b_43_n150_m15	50,21	6,71E+02
GKD-b_44_n150_m15	42,57	7,34E+02
GKD-b_45_n150_m15	36,21	8,13E+02
GKD-b_46_n150_m45	33,23	2,88E+03
GKD-b_47_n150_m45	23,49	2,73E+03
GKD-b_48_n150_m45	29,44	2,78E+03
GKD-b_49_n150_m45	34,22	2,91E+03
GKD-b_50_n150_m45	29,41	3,11E+03

También vamos a contruir una tabla de de resultados global que recoja los resultados medios de calidad y tiempo para todos los algoritmos considerados, agrupados por tamaño, y otra tabla en la que se muestre en total:

Algoritmo	**Tamaño**	**Desv**	**Tiempo**
Greedy	50	81,62	0
LSrandom	50	70,5	2,9
LSheur	50	72,3	2,6
AGG-uniforme	50	67,05	164
AGG-posicion	50	76,38	86,7
AGE-uniforme	50	66,96	250
AGE-posicion	50	72,59	158
AM-(10,1.0)	50	32,64	528
AM-(10,0.1)	50	50,8	329
AM-(10,0.1mej)	50	55,68	242
Greedy	100	74,48	3,8
LSrandom	100	62,35	29,5
LSheur	100	63,48	41,4
AGG-uniforme	100	54,81	423
AGG-posicion	100	68,17	109
AGE-uniforme	100	54,37	510
AGE-posicion	100	68,22	200
AM-(10,1.0)	100	20,94	2890
AM-(10,0.1)	100	35,61	838
AM-(10,0.1mej)	100	39,62	759
Greedy	150	69,15	22
LSrandom	150	56,06	94,6
LSheur	150	64,43	201
AGG-uniforme	150	50,08	772
AGG-posicion	150	66,89	173
AGE-uniforme	150	49,37	869
AGE-posicion	150	67,96	263
AM-(10,1.0)	150	20,45	8450
AM-(10,0.1)	150	36,37	1920
AM-(10,0.1mej)	150	37,73	1830

Cuadro 8: Resultados de los algoritmos Greedy, LSrandom, LSheur, AGG-uniforme,AGG-posicion,AGE-uniforme,AGE-posicion,AM-(10,1.0),AM-(10,0.1) y AM-(10,0.1mej) para los tamaños 50, 100 y 150.

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. ¿Qué operador de cruce es mejor?

Para responder a esta pregunta, vamos a observar por separado el AGG y el AGE. Para el AGG vemos que las medias de las desviaciones para el cruce uniforme es de $50,11$ y para el cruce de posición es de $61,61$, en la tabla en la que los algoritmos

Algoritmo	Desv	Tiempo
Greedy	67,06	7
LSrandom	55,71	35,3
LSheur	58,52	66,1
AGG-uniforme	50,11	405
AGG-posicion	61,61	111
AGE-uniforme	49,46	498
AGE-posicion	60,78	198
AM-(10,1.0)	20,88	3420
AM-(10,0.1)	33,78	897
AM-(10,0.1mej)	38,83	835

Cuadro 9: Media de Desviación y Tiempo para los algoritmos Greedy, LS-random, LSheur, AGG-uniforme,AGG-posicion,AGE-uniforme,AGE-posicion,AM-(10,1.0),AM-(10,0.1) y AM-(10,0.1mej).

se encuentran agrupados por tamaño también podemos comprobar que tanto para tamaño del problema, 50, 100 y 150 el AGG con cruce uniforme mejora al AGG con cruce de posición.

Si observamos el AGE, vemos que las medias de las desviaciones para el cruce uniforme es de $49,46$ y para el cruce de posición es de $60,78$ y del mismo modo para cada uno de los tamaños del problem, 50, 100 y 150 observamos que el cruce uniforme mejora al de posición.

Luego podemos decir que a teniendo en cuenta la desiación respecto a la solución óptima, el cruce uniforme con repración es mejor, algo que si tenemos en cuenta la diferencia de implementaciones no deberiá sorprendernos. Ambos tienen una característica en común y es que cuando se cruzan dos soluciones(en formato binario) las posiciones que tienen el mismo valor en una solución como en otra se dejan como están. La diferencia está en como gestiona cada operador el resto de posiciones. El cruce de posición lo hace totalmente aleatorio, luego no es solo que el cruce pueda no mejorar la solución, sino que ¡puede llegar a empeorarla!

Por otro lado, el cruce uniforme se basa de una heurística para reparar cada una de las soluciones lo que explica porque reduce las desviaciones respecto al otro operador de cruce.

Sin embargo, el cruce de posición, precisamente porque "no se complica.ª la hora de decididir que hcer con las posiciones que no coinciden, reduce en casi cuatro veces los tiempos de ejecución en el caso del AGG y casi tres veces en el caso del AGE.

2. ¿Qué algoritmo ofrece un mejor rendimiento, AGG o AGE?

Para realizar esta comparación vamos separarla por operadores de cruce.

En primer lugar, para el cruce uniforme, podemos ver que de media el AGG obtiene una desviación de $50,11$ frente a $49,46$ que obtiene el AGE, y en la tabla de tamaños también podemos ver que los resultados son prácticamente idénticos.

Respecto al cruce de posición, el AGG obtiene una media de $61,61$ el AGE de

60, 78 luego podemos ver que cnlos dos operadores decruce obtienen resltados muy similares. En la comparación por tamaños siguen sendo prácticaente idénticos tanto con tamaño del problema 0, 100 y 150.

En cuanto a los tiempo de AGG y AGE podemos ver que sonmuy similares siendo ligeramene sperior los dl AGE.

Rcordamos qe el AGG da un enfoque que permite mantener una alta diversidad genética y evitar que el algoritmo se estanque demasiado rápido. Es útil cuando el problema es complejo o cuando se sospecha que una amplia exploración del espacio de soluciones es necesaria.

En cambio el AGE, al ir tomando en torneo solo dos descendientes, de los cuales como mucho solo uno pasa a la siguiente generación tiene una convergencia más controlada y preserva soluciones buenas a lo largo del tiempo. Es útil en problemas donde la optimización incremental es preferible a una exploración amplia.

En este caso podemos que tanto un enfoque como otro nos llevan a resultados similares.

3. ¿Cuál de los tres algoritmos meméticos ofrece un mejor rendimiento?

Vamos de nuevo a fijarnos en los resultdos obtenidos. Podemos destacar las siguientes conclusiones:

- Cuando aplicamos la BLrandom a todas las soluciones mejora la calidad de las desviaciones frente a cuando solo se aplica un subconjunto reducido, como es un 10 %. Esto tiene sentido ya que estamos mejorando a la vez todas las soluciones(si no se encuentran en un óptmo local).
- El tiempo de ejecución cuando aplicamos la BLrandom a todas las soluciones aumenta bastant frente a los otros dos casos. Cuando se llama a la BLrandom se suman las evaluaciones que esta produce, luego podemos pensar que el aumento del tiempo se puede deber a que no realiza muchas evalaciones en cada llamada a la BLrandom, debido a que rapdamente llega a un optimo local, y prevalece el coste de realizar todas estas llamadas.
- En los dos casos en los qu solo se aplica la BLrandom a un 10 % de la población, obtenemos resultados similares en cuanto a desviación y tiempo. Luego podemos extraer cmo conclusón que ir modificando solo os mejores no tiene porque llevar a mejores soluciones. De hecho, el caso aleaorio es ligeramente mejor, lo que se puede a que al modifcar de manera aleatoria, cada vez se van modifiando comosomas distintos luego puedes llegar a soluciones que de otra manera no alcanzarías.

4. Comparación de los AM con el AGG con cruce uniforme

Ya hemos coentado que el AGG mantiene una alta diversidad genetica y esto puede hacer ue nos perdamos buenas soluciones que se conseguirían modificando ligeramente la solucion. Este es el objetivo de los algoritmos memeticos, introducir la BLrandom que permita de manera híbrida juntar el potencialquetiene AGG con la búsqueda local.

En vista de los resultados, podemos ver que efectivamente los AM mejoran con creces a los AGG, incluso reduciendo a lamitadla desviación.

5. Cmparación con los algoritmos de la Práctica 1.

Vamos a realizar también una breve comparación con los algoritmos Greedy, BLrandom y blheur comentadosen la anterior práctica. Podemos ver todos mejoran al algoritmo Greedy, que ya dijimos que en este problema es muy dificil que obtenga buenos resultados. Sin embargo los algoritmos genéticos con el cruce de posición, no mejoran a la búsqueda local, lo que nos da una idea de que este operador de cruce no es muy bueno.

El máximo potencial se alcanza al juntar la diversidad de soluciones que genera el AGG con u cruce más potente junto con la búsqueda loca, es decir, en los algoritmos memeticos.

Conclusión Final

Los resultados muestran que los algoritmos meméticos son la opción más efectiva para este problema, ya que combinan la exploración global de los algoritmos genéticos con la explotación local de las búsquedas locales. En particular, el uso del cruce uniforme con reparación y la aplicación estratégica de búsqueda local permite obtener soluciones de alta calidad, superando tanto a los algoritmos genéticos tradicionales como a los métodos clásicos como *Greedy* o búsqueda local pura. La elección del operador de cruce y del esquema de aplicación de la búsqueda local es clave para equilibrar la calidad de las soluciones y el coste computacional.