UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingeniería Informática y Telecomunicaciones

Práctica 1. 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

4 de abril 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 tres algoritmos: Greedy, BLheur y BLrandom. Estos tres algoritmos tienen en común una serie de aspectos del código que podemos ver en la carpeta common. Vamos air detallandolos poco a poco.

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.

```
1 typedef float tFitness;
2
3 typedef int tDomain;
4
5 typedef int tOption;
```

```
6
7 typedef tFitness tHeuristic;
8
9 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

Los tres algoritmos heredan de la clase MH que 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:

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

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

```
1 struct ResultMH {
2   tSolution solution;
3   tFitness fitness;
4   unsigned int evaluations;
5
6   ResultMH(tSolution &sol, tFitness fit, unsigned evals)
7       : solution(sol), fitness(fit), evaluations(evals) {}
8 };
```

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

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.

2.3.1. La factorización

En 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 llevarla a cabo, disponemos de una clase abstracta **SolutionFactoringInfo** a

partir de la cual hemos creado **MddpFactoringInfo**, que hereda de ella, en esta clase vamos a almacenar un array(info) con la información de la dispersión de cada elemento que se encuentra en solución, esto nos va a permitir no tener que calcular por completo la dispersión de una solución sino solo tener en cuenta la modificación de un elemento por otro, esencial en la Busqueda Local. Para que en todo momento esta información esté actualizada disponemos de los siguientes métodos:

1. generateFactoringInfo: Dada una solución, para cada elemento de la misma almacena en info su dispersión mediante llamadas al método calcularDelta. Devuelve un objeto de la clase SolutionFactoringInfo.

2. updateSolutionFactoringInfo: Es el método clave en la factorización, dada una solución, la información de la misma, el elemento que se va a eliminar y el elemento que se va a añadir, actualiza info de la siguiente manera: En la posición del array info donde estaba el elemento que se va a sustituir, se calcula la suma de las distancias al resto de elementos de solución, y para el resto de posiciones, se resta la distancia del elemento que se va a sustituir a la posición donde nos encontramos y se suma la distancia del elemento que se va a añadir a la posición donde nos encontramos.

Gracias a estos dos métodos, podemos crear un nuevo método fitness, mucho más eficiente, que calcule la dispersión que habría si se cambia un elemento i de solución por un elemento j que no se encuentra en solución, pasando de una eficiencia $O(n^2)$ a una efciencia de $O(n)$.

3. Pseudocódigo de cada algoritmo

Como ya hemos indicado, los tres algoritmos de los que vamos a hablar heredan de la clase MH e implmentan 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 Greedy

Para el algoritmo Greedy, en primer lugar hay que decir que se ha tenido en cuenta el hecho de que por la definición de la función objetivo, tanto si el numero m de elementos en la solución es 1 o 2, la dispersión mínima va a ser 0, luego en estos dos casos simplemenete cogemos los dos primeros elementos del total de n del problema. Para los casos en los que $m > 2$, cogemos los dos primeros elementos de menera aletoria y luego aplicamos la siguiente heurística: de entre los elementos que quedan sin seleccionar, se esocoge el que de lugar a una menor dispersión en el nuevo conjunto de seleccionados. Esto se hace en los $m - 2$ pasos siguientes.

Pseudocódigo función optimize:

Pseudocódigo función optimize

```
Funcion optimize(problem, maxevals) devuelve ResultMH
  Definir ListaValores como un vector con valores enteros desde
0 a n-1
  Definir Solucion como un vector vacío
  Definir ProblemaReal como problem convertido a Mddp
  Si m = 1 Entonces:
    Agregar primer elemento de ListaValores a Solucion
    fitness ← fitness(ProblemaReal, Solucion)
    return ResultMH(Solucion, fitness, 1)
  FinSi
  Si m = 2 Entonces:
    Agregar los dos primeros elementos de ListaValores a
Solucion
    fitness ← fitness(ProblemaReal, Solucion)
    return ResultMH(Solucion, fitness, 1)
  FinSi
  indice ← SeleccionarAleatorio(0, n-1)
  Agregar ListaValores[indice] a Solucion
  Eliminar ListaValores[indice]
  indice ← SeleccionarAleatorio(0, n-2)
  Agregar ListaValores[indice] a Solucion
  Eliminar ListaValores[indice]
  m ← m-2
  Mientras m >0 Hacer:
    Definir CopiaSolucion como una copia de Solucion
    MejorIndice ← 0
    Agregar ListaValores[0] a CopiaSolucion
    fitness ← fitness(ProblemaReal, CopiaSolucion)
    Para i desde 1 hasta Longitud(ListaValores)-1 Hacer:
      Eliminar UltimoElemento de CopiaSolucion
      Agregar ListaValores[i] a CopiaSolucion
      newfitness ← fitness(ProblemaReal, CopiaSolucion)
      Si newfitness <fitness Entonces:
        MejorIndice ← i
        fitness ← newfitness
      FinSi
    FinPara
    Agregar ListaValores[MejorIndice] a Solucion
    Eliminar ListaValores[MejorIndice]
    m ← m-1
  FinMientras
  return ResultMH(Solucion, fitness, 1)
FinFuncion
```

3.2. Busqueda Local

Los algoritmos de búsqueda local se basan en que partiendo de una solución del problema, buscan una solución vecina que mejore la actual, y así hasta que se llegue a un número máximo de iteraciones que se establece como umbral o se alcanze un óptimo(que puede ser local). Una solución vecina consiste en que dado el vector de la solución, se intercambia un elemento i del mismo por un elemento j de los que no están en la solución. Para hacer estos cambios de manera más eficiente se usa la factorización explicada en la sección 2.3.1.
Para resolver este problema hemos creado dos algoritmos de búsqueda local, que se diferencian en la manera en la que exploran el entorno:

1. En orden totalmente aleatorio(BLrandom)
2. En función de una heurística(BLheur)

3.2.1. Algoritmo BLheur

Este algoritmo implementa la búsqueda local realizando la exploración del entorno de la siguiente manera: Usando la factorización, comprueba que movimiento, i.e. sustituir un elemento i de la solución por un elemento j de los que no se hallan en ella, da lugar a una menor dispersión, una vez exploradas todas las opciones, realiza la sustirtución correspondiente y actualiza la instancia de la clase SolutionFactoringInfo. Puede ser que no haya ningún intercambio que mejore la solución actual, en cuyo caso se finaliza la ejecución, al igual que si se alcanza el numero maximo de evaluaciones.

En el pseudocódigo se puede observar que primero inicializamos una solución de manera aleatoria, y luego mientras haya mejora y no se llegue al numero maximo de evaluaciones buscamos el intercambio que reduzca más la dispersión de la solución actual.

Pseudocódigo función optimize

```
Funcion optimize(problem, maxevals) devuelve ResultMH
  assert(maxevals >0)
  Definir realproblem como problem convertido a Mddp
  Definir seleccionados como un vector vacío
  Definir no_seleccionados como un vector vacío
  Definir aux como un vector de tamaño
realproblem.getProblemSize() con valores de 0 a n-1
  aux ← MezclarAleatoriamente(aux)
  Para i desde 0 hasta tamaño de aux - 1 hacer:
    Si i <realproblem.getSolutionSize() Entonces
      Agregar aux[i] a seleccionados
    SiNo
      Agregar aux[i] a no_seleccionados
    FinSi
  FinPara
  Definir info como
```

```
realproblem.generateFactoringInfo(seleccionados)
  mejor_fitness ← realproblem.fitness(seleccionados)
  evaluaciones ← 1
  mejora ← Verdadero
  Mientras evaluaciones ≤ maxevals y mejora Hacer
    mejora ← Falso
    Definir pos_in, pos_out
    mejor_fitness_aux ← mejor_fitness
    Para i desde 0 hasta tamaño de seleccionados - 1 hacer:
      Para j desde 0 hasta tamaño de no_seleccionados - 1 hacer:
        elem_in ← no_seleccionados[j]
        nuevo_fitness ← realproblem.fitness(seleccionados, info,
i, elem_in)
        Si nuevo_fitness <mejor_fitness_aux Entonces
          mejor_fitness_aux ← nuevo_fitness
          pos_out ← i
          pos_in ← j
          mejora ← Verdadero
        FinSi
      FinPara
    FinPara
    Si mejora Entonces
      evaluaciones ← evaluaciones + 1
      mejor_fitness ← mejor_fitness_aux
      elem_out ← seleccionados[pos_out]
      elem_in ← no_seleccionados[pos_in]
      realproblem.updateSolutionFactoringInfo(info,
seleccionados, pos_out, elem_in)
      seleccionados[pos_out] ← elem_in
      no_seleccionados[pos_in] ← elem_out
    FinSi
  FinMientras
  return ResultMH(seleccionados, mejor_fitness, evaluaciones)
FinFuncion
```

3.2.2. Algoritmo BLrandom

Este algoritmo implementa la búsqueda local realizando la exploración del entorno de la siguiente manera: Usando la factorización, escoge de manera aleatoria de entre todos los posibles el primer movimiento, i.e. sustituir un elemento i de la solución por un elemento j de los que no se hallan en ella, que da lugar a una menor dispersión, realiza la sustirtución correspondiente y actualiza la instancia de la clase Solution-FactoringInfo. Puede ser que no haya ningún intercambio que mejore la solución actual, en cuyo caso se finaliza la ejecución, al igual que si se alcanza el numero maximo de evaluaciones.

En el pseudocódigo se puede observar que primero inicializamos una solución de

manera aleatoria, y luego mientras haya mejora y no se llegue al numero maximo de evaluaciones creamos una matriz con todas las posibilidades de intercambio, con Shuffle la desordenamos de manera aleatoria, y buscamos el primer intercambio que reduzca la dispersión de la solución actual.

Pseudocódigo función optimize

```
Funcion optimize(problem, maxevals) devuelve ResultMH
  assert(maxevals >0)
  Definir ProblemaReal como problem convertido a Mddp
  Definir Seleccionados como un vector vacío
  Definir NoSeleccionados como un vector vacío
  Definir Aux como un vector con valores enteros desde 0 a
ProblemSize - 1
  Random.shuffle(Aux)
  Para i desde 0 hasta ProblemSize - 1 Hacer:
    Si i <SolutionSize Entonces:
      Agregar Aux[i] a Seleccionados
    Sino
      Agregar Aux[i] a NoSeleccionados
    FinSi
  FinPara
  Definir Info como la información factorizada de ProblemaReal
usando Seleccionados
  fitness ← fitness(ProblemaReal, Seleccionados)
  evaluaciones ← 1
  mejora ← Verdadero
  Mientras evaluaciones ≤ maxevals y mejora Hacer:
    mejora ← Falso
    Definir Posibilidades como una matriz de pares vacía
    Para i desde 0 hasta Longitud(Seleccionados) - 1 Hacer:
      Definir Aux como un vector vacío
      Para j desde 0 hasta Longitud(NoSeleccionados) - 1 Hacer:
        Agregar (i, j) a Aux
      FinPara
      Random.shuffle(Aux)
      Agregar Aux a Posibilidades
    FinPara
    Random.shuffle(Posibilidades)
    Para i desde 0 hasta Longitud(Posibilidades) - 1 Hacer:
      Para j desde 0 hasta Longitud(Posibilidades[i]) - 1 Hacer:
        pos_out ← Posibilidades[i][j].first
        pos_in ← Posibilidades[i][j].second
        nuevo_fitness ← fitness(ProblemaReal, Seleccionados,
Info, NoSeleccionados[pos_in])
        Si nuevo_fitness <fitness Entonces:
          Actualizar la información factorizada con el
```

```
intercambio
          Intercambiar Seleccionados[pos_out] con
NoSeleccionados[pos_in]
          fitness ← nuevo_fitness
          mejora ← Verdadero
        FinSi
        Si mejora Entonces romper bucle
      FinPara
      Si mejora Entonces romper bucle
    FinPara
    evaluaciones ← evaluaciones + 1
  FinMientras
  return ResultMH(Seleccionados, fitness, evaluaciones)
FinFuncion
```

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
   - 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).

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 tres 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(Greedy, BLheur, BLrandom) donde se recogen los resultados de la ejecución de dicho algoritmo al conjunto de casos del problema:

Cuadro 1: Resultados del algoritmo Greedy

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0,00	0,00E+00
GKD-b_2_n25_m2	0,00	0,00E+00

Caso	Desv	Tiempo
GKD-b_3_n25_m2	0,00	0,00E+00
GKD-b_4_n25_m2	0,00	0,00E+00
GKD-b_5_n25_m2	0,00	0,00E+00
GKD-b_6_n25_m7	76,65	0,00E+00
GKD-b_7_n25_m7	81,30	0,00E+00
GKD-b_8_n25_m7	74,45	0,00E+00
GKD-b_9_n25_m7	77,27	0,00E+00
GKD-b_10_n25_m7	66,15	0,00E+00
GKD-b_11_n50_m5	93,21	0,00E+00
GKD-b_12_n50_m5	93,16	0,00E+00
GKD-b_13_n50_m5	90,85	0,00E+00
GKD-b_14_n50_m5	94,06	0,00E+00
GKD-b_15_n50_m5	86,41	0,00E+00
GKD-b_16_n50_m15	83,17	0,00E+00
GKD-b_17_n50_m15	69,73	0,00E+00
GKD-b_18_n50_m15	70,62	0,00E+00
GKD-b_19_n50_m15	73,17	0,00E+00
GKD-b_20_n50_m15	61,89	0,00E+00
GKD-b_21_n100_m10	76,04	0,00E+00
GKD-b_22_n100_m10	81,55	0,00E+00
GKD-b_23_n100_m10	82,25	0,00E+00
GKD-b_24_n100_m10	89,21	0,00E+00
GKD-b_25_n100_m10	77,33	0,00E+00
GKD-b_26_n100_m30	62,38	9,00E+00
GKD-b_27_n100_m30	70,37	7,00E+00
GKD-b_28_n100_m30	77,54	7,00E+00
GKD-b_29_n100_m30	62,20	6,00E+00
GKD-b_30_n100_m30	65,92	9,00E+00
GKD-b_31_n125_m12	87,14	1,00E+00
GKD-b_32_n125_m12	73,78	1,00E+00
GKD-b_33_n125_m12	79,77	1,00E+00
GKD-b_34_n125_m12	78,94	1,00E+00
GKD-b_35_n125_m12	79,10	1,00E+00
GKD-b_36_n125_m37	69,99	1,60E+01
GKD-b_37_n125_m37	69,50	1,60E+01
GKD-b_38_n125_m37	66,53	1,80E+01
GKD-b_39_n125_m37	62,38	1,90E+01
GKD-b_40_n125_m37	57,68	1,80E+01
GKD-b_41_n150_m15	75,26	2,00E+00
GKD-b_42_n150_m15	78,99	2,00E+00
GKD-b_43_n150_m15	78,94	2,00E+00
GKD-b_44_n150_m15	76,57	2,00E+00
GKD-b_45_n150_m15	74,28	2,00E+00
GKD-b_46_n150_m45	63,42	5,40E+01
GKD-b_47_n150_m45	63,10	4,10E+01
GKD-b_48_n150_m45	61,79	3,80E+01

Caso	Desv	Tiempo
GKD-b_49_n150_m45	62,05	3,90E+01
GKD-b_50_n150_m45	57,18	3,80E+01

Cuadro 2: Resultados del algoritmo Blheur

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0,00	0,00E+00
GKD-b_2_n25_m2	0,00	0,00E+00
GKD-b_3_n25_m2	0,00	0,00E+00
GKD-b_4_n25_m2	0,00	0,00E+00
GKD-b_5_n25_m2	0,00	0,00E+00
GKD-b_6_n25_m7	67,09	0,00E+00
GKD-b_7_n25_m7	56,87	0,00E+00
GKD-b_8_n25_m7	54,85	0,00E+00
GKD-b_9_n25_m7	54,46	0,00E+00
GKD-b_10_n25_m7	37,82	0,00E+00
GKD-b_11_n50_m5	87,86	0,00E+00
GKD-b_12_n50_m5	86,62	0,00E+00
GKD-b_13_n50_m5	85,14	0,00E+00
GKD-b_14_n50_m5	91,26	0,00E+00
GKD-b_15_n50_m5	77,98	0,00E+00
GKD-b_16_n50_m15	63,75	5,00E+00
GKD-b_17_n50_m15	54,16	7,00E+00
GKD-b_18_n50_m15	54,45	6,00E+00
GKD-b_19_n50_m15	65,83	4,00E+00
GKD-b_20_n50_m15	55,98	4,00E+00
GKD-b_21_n100_m10	67,36	5,00E+00
GKD-b_22_n100_m10	69,16	4,00E+00
GKD-b_23_n100_m10	68,68	4,00E+00
GKD-b_24_n100_m10	76,15	4,00E+00
GKD-b_25_n100_m10	51,39	4,00E+00
GKD-b_26_n100_m30	52,47	7,70E+01
GKD-b_27_n100_m30	66,65	8,00E+01
GKD-b_28_n100_m30	66,51	8,70E+01
GKD-b_29_n100_m30	55,42	7,10E+01
GKD-b_30_n100_m30	60,99	7,80E+01
GKD-b_31_n125_m12	77,22	1,40E+01
GKD-b_32_n125_m12	62,66	9,00E+00
GKD-b_33_n125_m12	63,49	9,00E+00
GKD-b_34_n125_m12	62,74	9,00E+00
GKD-b_35_n125_m12	62,33	8,00E+00
GKD-b_36_n125_m37	68,09	1,36E+02
GKD-b_37_n125_m37	61,40	2,02E+02
GKD-b_38_n125_m37	66,07	1,66E+02
GKD-b_39_n125_m37	66,95	1,38E+02
GKD-b_40_n125_m37	61,68	1,61E+02

Caso	Desv	Tiempo
GKD-b_41_n150_m15	67,57	2,00E+01
GKD-b_42_n150_m15	72,79	2,00E+01
GKD-b_43_n150_m15	74,65	2,60E+01
GKD-b_44_n150_m15	64,02	2,40E+01
GKD-b_45_n150_m15	59,16	2,20E+01
GKD-b_46_n150_m45	67,52	3,37E+02
GKD-b_47_n150_m45	56,75	4,30E+02
GKD-b_48_n150_m45	55,66	4,22E+02
GKD-b_49_n150_m45	63,20	3,45E+02
GKD-b_50_n150_m45	63,00	3,65E+02

Cuadro 3: Resultados del algoritmo Blrandom

Caso	Desv	Tiempo
GKD-b_1_n25_m2	0,00	0,00E+00
GKD-b_2_n25_m2	0,00	0,00E+00
GKD-b_3_n25_m2	0,00	0,00E+00
GKD-b_4_n25_m2	0,00	0,00E+00
GKD-b_5_n25_m2	0,00	0,00E+00
GKD-b_6_n25_m7	64,97	0,00E+00
GKD-b_7_n25_m7	58,91	0,00E+00
GKD-b_8_n25_m7	54,68	0,00E+00
GKD-b_9_n25_m7	54,04	0,00E+00
GKD-b_10_n25_m7	46,77	0,00E+00
GKD-b_11_n50_m5	84,87	0,00E+00
GKD-b_12_n50_m5	84,71	0,00E+00
GKD-b_13_n50_m5	80,15	0,00E+00
GKD-b_14_n50_m5	92,08	0,00E+00
GKD-b_15_n50_m5	81,12	0,00E+00
GKD-b_16_n50_m15	68,00	5,00E+00
GKD-b_17_n50_m15	55,04	7,00E+00
GKD-b_18_n50_m15	53,72	6,00E+00
GKD-b_19_n50_m15	60,97	6,00E+00
GKD-b_20_n50_m15	44,33	5,00E+00
GKD-b_21_n100_m10	66,11	8,00E+00
GKD-b_22_n100_m10	65,00	5,00E+00
GKD-b_23_n100_m10	58,01	5,00E+00
GKD-b_24_n100_m10	80,14	4,00E+00
GKD-b_25_n100_m10	55,38	4,00E+00
GKD-b_26_n100_m30	52,91	4,50E+01
GKD-b_27_n100_m30	57,88	5,20E+01
GKD-b_28_n100_m30	62,88	6,90E+01
GKD-b_29_n100_m30	57,80	6,00E+01
GKD-b_30_n100_m30	67,39	4,30E+01
GKD-b_31_n125_m12	79,40	1,00E+01
GKD-b_32_n125_m12	62,32	7,00E+00

Caso	Desv	Tiempo
GKD-b_33_n125_m12	62,29	1,00E+01
GKD-b_34_n125_m12	65,74	1,10E+01
GKD-b_35_n125_m12	64,48	1,20E+01
GKD-b_36_n125_m37	54,65	1,04E+02
GKD-b_37_n125_m37	53,79	8,10E+01
GKD-b_38_n125_m37	61,76	7,90E+01
GKD-b_39_n125_m37	62,68	7,50E+01
GKD-b_40_n125_m37	49,86	1,07E+02
GKD-b_41_n150_m15	62,75	1,80E+01
GKD-b_42_n150_m15	62,47	1,80E+01
GKD-b_43_n150_m15	51,88	2,40E+01
GKD-b_44_n150_m15	55,45	2,00E+01
GKD-b_45_n150_m15	59,65	1,70E+01
GKD-b_46_n150_m45	59,28	1,44E+02
GKD-b_47_n150_m45	44,15	1,92E+02
GKD-b_48_n150_m45	51,21	1,90E+02
GKD-b_49_n150_m45	57,33	1,51E+02
GKD-b_50_n150_m45	56,40	1,72E+02

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	85,65	4,90
LSrandom	50	76,76	4,90
LSheur	50	64,38	5,00
Greedy	100	78,46	16,26
LSrandom	100	61,87	27,32
LSheur	100	68,13	5,60
Greedy	150	76,18	207,68
LSrandom	150	56,39	200,00
LSheur	150	63,16	222,10

Cuadro 4: Resultados de los algoritmos Greedy, LSrandom y LSheur para los tamaños 50, 100 y 150.

Algoritmo	Desv	Tiempo
Greedy	80,43	76,28
LSrandom	64,34	77,74
LSheur	65,22	77,23

Cuadro 5: Media de Desviación y Tiempo para los algoritmos Greedy, LSrandom y LSheur.

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. Para ayudarnos

al análisis, además vamos a mostrar un boxplot de las desviaciones de cada uno de los algoritmos. Recordemos que la caja central abarca desde el percentil 25 (Q1) hasta el percentil 75 (Q3). La línea dentro de la caja indica la Mediana. Los bigotes muestran los valores dentro de 1.5 veces el IQR. Por último los puntos fuera de los bigotes son valores atípicos.

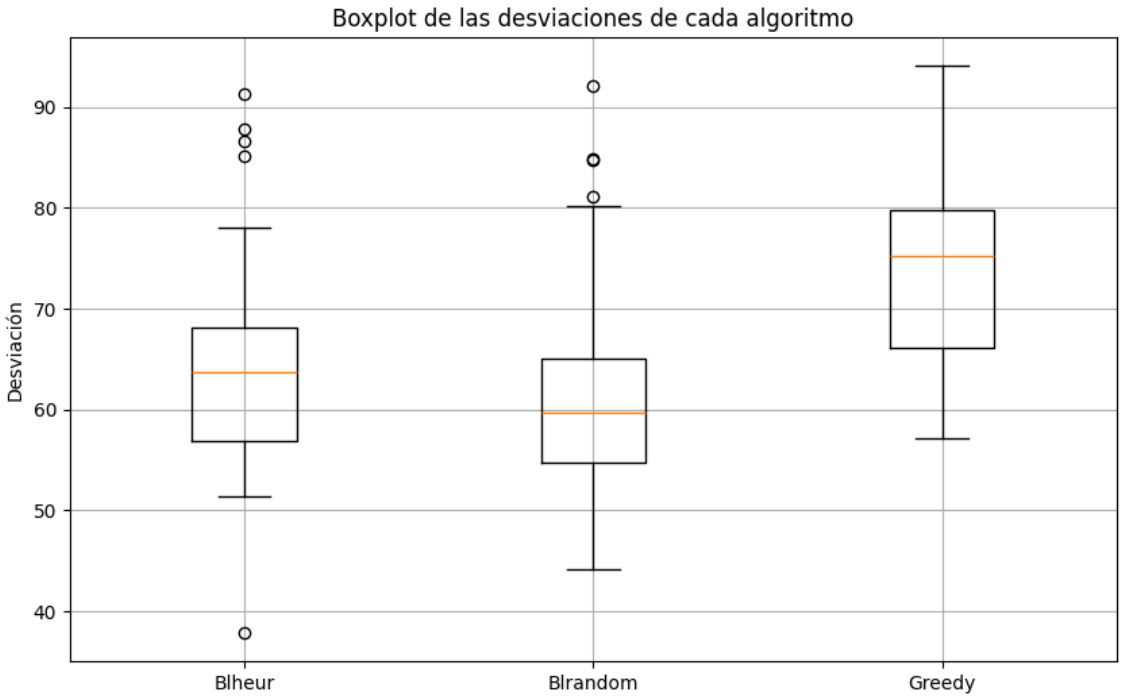

Figura 2: Boxplot de las desviaciones de los algoritmos Blheur, Blrandom y Greedy.

Como podemos ver en las tablas y en los boxplots, el algoritmo Greedy obtiene peores reultados que los algoritmos de Búsqueda Local. Además el algoritmo BLrandom tiene resultados más dispersos que BLheur, lo que tiene sentido dada la naturaleza aleatoria del mismo.

Recordamos que el algoritmo Greedy en cada paso busca el elemento que da lugar a una menor dispersión entre los no seleccionados, y repetimos hasta tener una solución completa, la cual devolvemos.

Este método, aunque a priori pueda parecer bueno, toma decisiones locales en cada paso sin considerar el impacto global. Además se ve afectado por la elección aleatoria inicial, que restringe sus posibilidades. Veamos un ejemplo:

Vamos a considerar $n = 5$ y $m = 4$ y la siguiente matriz de distancias D:

$$D = \begin{bmatrix} 0 & 2 & 9 & 10 & 7 \\ 2 & 0 & 6 & 5 & 8 \\ 9 & 6 & 0 & 4 & 3 \\ 10 & 5 & 4 & 0 & 2 \\ 7 & 8 & 3 & 2 & 0 \end{bmatrix}$$

Vamos a suponer que de manera aleatoria elige como dos primeros elementos e_2 y e_4:

Cálculo inicial de $\Delta(v)$:

$$\Delta(e_2) = d_{2,4} = 5, \quad \Delta(e_4) = d_{4,2} = 5.$$

$$\text{diff}(S') = 5 - 5 = 0.$$

Ahora vamos a escoger el tercer elemento de acuerdo con la metodologia Greedy, es decir, minimizando diff(S').

- Si añadimos e_1:

$$\Delta(e_1) = d_{1,2} + d_{1,4} = 2 + 10 = 12,$$
$$\Delta(e_2) = 7, \quad \Delta(e_4) = 15.$$
$$\text{diff}(S') = 15 - 7 = 8.$$

- Si añadimos e_3:

$$\Delta(e_3) = d_{3,2} + d_{3,4} = 6 + 4 = 10,$$
$$\Delta(e_2) = 11, \quad \Delta(e_4) = 9.$$
$$\text{diff}(S') = 11 - 9 = 2.$$

Luego elegimos e_3, y tenemos $S' = \{e_2, e_3, e_4\}$.

Para el cuarto elemento realizamos operaciones análogas con los elementos no seleccionados e_1 y e_5 y el Greedy elige e_5:

$$S' = \{e_2, e_3, e_4, e_5\}, \quad \text{diff}(S') = 8.$$

Sin embargo, la siguente solución:

$$S^* = \{e_1, e_3, e_4, e_5\},$$

da lugar a la siguiente dispersión:

$$\text{diff}(S^*) = 19 - 13 = 6.$$

Luego con un ejemplo sencillo podemos ver que el Greedy no alcanza la solución óptima,, lo que es coherente con los resultados obtenidos.

Por otro lado, los algoritmos de Búsqueda Local precisamente lo que buscan es minimizar el error que pueda ocasionar tener una solución y que haya una muy parecida que sea mucho mejor, es decir, cambiando algunos elementos de la solución esta mejore, sin embargo, como vemos en los resultados esto tampoco nos permite llegar a la solución óptima y evitar que se pierda en óptimos locales, aunque si mejora al algoritmo Greedy.

También hay que tener en cuenta, que el Greedy solo se ejecuta una vez, mientras que las Búsquedas Locales tienen un número máximo de evaluaciones(de intercambios) de 100000, aunque en ninguno de los casos probados se llegan a acercar a dicha cifra.

Para terminar este análisis me gustaría hablar de la comparación entre la Búsqueda Local random, en la que la exploración del entorno se hace de manera aleatoria, y la Búsqueda Local heurística, en la que se escoge como intercambio aquel quu de lugar a una mejor dipersióm, en el caso de que hubiese alguno que cumpla tal condición. Cabría esperar que esta segunda opción de lugar a mejores resultados, sin embargo podemos ver que en tamaños mayores, con la exploración aleatoria se obtienen mejores resultados, lo que nos hace ver que a pesar de implementar esta heurística se sigue quedando en óptimos locales.

Otro apunte interesante es que los tres algoritmos, de manera general, conforme se aumenta el tamaño de los problemas, generan una mejor solución.

Por último, respecto a los tiempos de cada algoritmo, podemos ver que los tres son parecidos, así que no nos podemos basar en esta métrica para determinar que algoritmo es mejor o peor.