METAHEURÍSTICAS

PRÁCTICA 3.a:

Búsquedas por Trayectorias para el
Problema de la Mínima Dispersión Diferencial

UNIVERSIDAD
DE GRANADA

Ignacio Yuste López, 54141533Q

ignacioyustel@correo.ugr.es

Grupo 2

1. Índice

1. Índice....2
2. Descripción del problema....3
3. Descripción de los algoritmos empleados....4
3.1. Algoritmo Greedy....4
3.2. Algoritmo Búsqueda Local Mejor....6
3.3. Algoritmos Genéticos....9
3.3.1. Algoritmo Genético Generacional....9
3.3.2. Algoritmo Genético Estacionario....10
3.3.3. Algoritmo de cruce uniforme....11
3.3.4. Algoritmo de cruce de posición....12
3.5. Algoritmo Memético....13
3.6. Algoritmo de Enfriamiento Simulado (ES)....14
3.7. Algoritmo de búsqueda multiarranque básica (BMB)....16
3.8. Algoritmos de búsqueda local reiterativos basados en óptimos (ILS)....16
4. Procedimiento para desarrollar la practica....17
4.1. Manual de uso....18
5. Experimentos y análisis de resultados....19
5.1. Análisis de BLM y Greed (práctica 1)....20
5.2. Análisis de algoritmos basados en poblaciones (práctica 2)....23
5.2.1. Análisis de algoritmos genéticos....23
5.2.2. Análisis de los Algoritmos Meméticos....26
5.2.3. Algoritmos Genéticos vs Meméticos....29
5.3. Análisis de algoritmos basados en trayectorias (práctica 3)....30
5.3.1. Comparación de algoritmos de búsqueda local....30
5.3.2. Comparación de algoritmos de trayectorias múltiples....31
6. Bibliografía....34

2. Descripción del problema.

El problema elegido para esta práctica es el Problema de la Mínima Dispersión Diferencial. Este es un problema de optimización combinatoria, aparentemente sencillo, que resulta ser NP-Completo.

El problema consiste en seleccionar un subconjunto *Sel* de elementos de un conjunto inicial *S* tal que la dispersión entre los elementos de dicho subconjunto sea la mínima posible. Para ello, además del número de elementos del subconjunto, disponemos de la distancia entre todos los elementos en una matriz cuadrada tal que la distancia entre *i* y *j* es *d(i,j)*.

Para esta práctica calcularemos la dispersión como la diferencia entre los valores extremos del subconjunto *Sel*:

$$diff\left(S\right)=max_{\left(u\in S\right)}\Delta\left(u\right)-min_{\left(v\in S\right)}\Delta\left(v\right)$$

Donde para cada punto elegido *v* se calcula Δ(v) como la suma de las distancias de ese punto al resto del subconjunto.

$$\Delta\left(v\right)=\Sigma_{\left(u\in S\right)}d_{uv}$$

Por tanto, nuestro objetivo es minimizar la medida de dispersión

$$S^{✪}=argmin_{\left(S\subseteq V_m\right)}diff\left(S\right)$$

3. Descripción de los algoritmos empleados.

Para todos los algoritmos aquí explicados partimos de las mismas bases:

- *solucion*: Conjunto solución. Es un subconjunto del espacio de búsqueda que cumple las condiciones de tener un tamaño *m* y no tener ningún elemento repetido, buscando tener la mínima dispersión posible entre ellos.
- *CL(Para Greedy y BLM)*: Conjunto de espacio de búsqueda. Contiene todos los índices (las localizaciones dicho de otra forma) entre los que tenemos que buscar los *m* elementos con la mínima dispersión posible.
- *Población (Para algoritmos basados en poblaciones)*: Conjunto de espacio de búsqueda. Consiste en una matriz en la cual cada fila es un cromosoma de una población. Un cromosoma consiste en un vector de N elementos (siendo N el número total de posibilidades de localización) de 0s y 1s donde 1 significa que ese cromosoma contiene esa localización. A cada uno de estos valores los llamamos gen del cromosoma. Cada cromosoma tiene *m* genes "activos", es decir, con valor 1.
- *datos*: Matriz cuadrada que contiene las distancias entre todas las localizaciones. En mi implementación cada distancia está duplicada, ya que se trata de una matriz simétrica por comodidad de implementación.
- *m*: Número de elementos cuya dispersión debe ser lo mínima posible

3.1. Algoritmo Greedy.

La mayor virtud del algoritmo *Greedy* o *Voraz* es su simpleza de implementación y sus tiempos de ejecución.

Partiendo de un subconjunto solución *Sel* con un solo valor elegido aleatoriamente, comprueba cual del resto de elementos es el que menos dispersión añade al sumarse al subconjunto actual. Si es el mínimo entre todos en el espacio de búsqueda, lo añade y vuelve a empezar con el nuevo subconjunto.

```
Algoritmo Greedy(m, datos):

    solucion <- Null
    CL <- datos.indices

    v0 <- SeleccionAleatoria(CL)
    solucion <- solucion U v0
    CL <- CL \ v0

    while |solucion| < m do

        u <- funcionHeuristica(solucion, CL, datos)
        solucion <- solucion U u
        CL <- CL \ u

    end while

    return solucion
```

En el pseudocódigo vemos cómo se comienza con un conjunto *solucion* vacío y un conjunto *CL* el cual será nuestro espacio de búsqueda. El conjunto *CL* contiene los índices del conjunto de datos, es decir, los números comprendidos entre 0 y el número de filas de la matriz de datos.

Se elige un índice aleatoriamente y se añade al subconjunto *solucion* y se elimina del espacio de búsqueda.

Buscamos a través de *funcionHeuristica* el mejor elemento a añadir de entre todos los del espacio de búsqueda, lo añadimos a *solucion* y lo eliminamos de *CL.* Iteramos este último paso hasta conseguir un subconjunto solución del tamaño *m* deseado.

```
funcionHeuristica(solucion, RCL, datos):

    for(u in RCL) do:

        for (v in solucion) do:
            heuristico(u) <- heuristico(u) + datos(u, v)

        end for

        for (v in solucion) do:
            heuristico(v) <- heuristico(u) + datos(u, v)

        end for

        g(u) <- max(heuristico(u), max(heuritico(v)) - min(heuristico(u), min(heuristico(v))

    end for

    return u <- min(g(u))
```

La función *funcionHeuristica* es a la vez la función de búsqueda y la función objetivo. Para cada elemento *u* del espacio de búsqueda calcula su distancia a los elementos del subconjunto *solucion* y se almacena como su valor heurístico con respecto a dicho subconjunto. Para cada elemento *v* en *solucion* se almacena el valor heurístico de *u* más la distancia de *u* a *v.* Por último, guarda el verdadero valor heurístico de *u* como *g(u)*, siendo este la diferencia antes expuesta. Por último, busca y devuelve la *u* perteneciente al espacio de búsqueda cuyo valor heurístico es el menor.

Por tanto, en cada iteración del bucle principal añadimos un nuevo valor a *solucion* siendo este el mejor entre el espacio de búsqueda, el cual también se reduce en cada iteración ya que no comprende los valores ya añadidos a *solucion*.

3.2. Algoritmo Búsqueda Local Mejor.

Este algoritmo se basa en explorar vecinos y comprobar si estos son mejores que la solución actual.

En este caso partimos de un conjunto *solucion* de tamaño *m* con elementos seleccionados aleatoriamente. Esta es nuestra *solucion* original. Estos *m* elementos se descartan del espacio de búsqueda.

El pseudocódigo del algoritmo es el siguiente:

```
Algoritmo BLM(m, datos):

    for(i in m) do:
        sel(i) <- Random(0, datos.indices)

    end for

    for(i in datos.filas) do:
        for(j in datos.columnas) do:
            if (i!=j):
                index(i) <- make_pair(i, j)
        end for
    end for

    contador <- 0
    seIntercambia <- true

    while (contador < 100000 and seIntercambia) do:

        mezcla(index)
        seIntercambia <- false

        for(i in index and !seIntercambia) do:

            contador <- contador + 1

            inSel <- index(i).first in solucion
            outSel <- index(i).second not in solucion

            if(inSel and outSel):
                seIntercambia <- Int(solucion, index(i), datos)

        end for

    end while

    return solucion

end
```

El vecindario (así llamaremos al espacio de búsqueda en este caso) consiste en el conjunto de intercambios posibles, siendo un intercambio cambiar un elemento $v \in solucion$ por un elemento $u \in CL$. Los intercambios son representados como una pareja de índices que intercambiar. Para este caso definimos un nuevo vecindario en desordenando (mezclando) el conjunto de parejas. Cada orden distinto es un nuevo vecindario.

Lo interesante del algoritmo es su procedimiento de búsqueda, ya que este itera mientras encontremos un intercambio válido en cada vecindario. Si encontramos ese intercambio, crea un nuevo vecindario y vuelve a empezar. En el momento en el que se explore un vecindario al completo sin encontrar un intercambio aceptable el algoritmo termina. Sin embargo, hemos definido un tope de iteraciones ya que para algunos conjuntos de datos este procedimiento puede ser muy costoso temporalmente.

Para comprobar si un intercambio es aceptable primero comprobamos que el vecino a explorar contiene un elemento del conjunto *solucion* y otro elemento que no está en él. Si cumple esta simple condición, llamamos a la función *Int*, la cual comprueba si el intercambio es aceptable, en cuyo caso procede a hacer el intercambio y confirma si se hace el intercambio o no a la función que la llama.

Aquí vemos el pseudocódigo de la función *Int*:

```
Int(solucion, intercambio, datos):

    diffOriginal <- max(coste(u) in solucion) - min(coste(u) in solucion)
    distancia_v <- 0
    sel_int <- solucion

    for(i in sel_int) do:

        distancia_v <- distancia_v + datos(intercambio.second, sel_int(i))
        coste(sel_int(i)) <- sel_int(i) - datos(intercambio.first, sel_int(i))
        coste(sel_int(i)) <- sel_int(i) + datos(intercambio.second, sel_int(i))

    end for

    diferencia_coste <- max(distancia_v, max(coste(u) in sel_int))
                        - min(distancia_v, min(coste(u) in sel_int))
                        - diffOriginal

    if(diferencia_coste < 0):
        intercambia(sel, intercambio.first, intercambio.second)
        return true

    return false

end
```

La condición de aceptación de la que hablábamos antes se trata de que la diferencia del coste con el intercambio y el coste antes de él sea negativa, es decir, que el coste del conjunto *solucion* después de intercambiar es menor que antes del intercambio.

Ya que el vecindario es demasiado grande calcular el coste de todas las soluciones implica calcular en cada iteración la distancia de cada uno de los *m* elementos de *solucion* al resto. Por este motivo se opta por una estrategia con una factorización del cálculo del coste.

Esta factorización consiste en restarle a los elementos de *solucion* la distancia al primer elemento del intercambio y sumarle las del segundo:

$$solucion = s_1, \dots, i, \dots, s_m \rightarrow solucion' = solucion - (i) + (j) \rightarrow solucion' = s_1, \dots, j, \dots, s_m$$

Después se calcula la diferencia entre los costes *solucion* y *solucion'* y si esta es negativa se hace el intercambio.

Por último, mencionar el procedimiento seguido para mezclar el índice de intercambios y crear los vecindarios. A continuación se muestra el pseudocódigo de la función *mezcla*:

```
mezcla (index):

    for(i in index) do:

        random <- Random(0, |index|)
        aux <- index(i)
        index(i) <- index(random)
        index(random) <- aux

    end for

    return index

end
```

Es un algoritmo sencillo. Para cada valor de *index* se selecciona otro valor aleatoriamente y se intercambian.

3.3. Algoritmos Genéticos

Estos algoritmos hacen una búsqueda basada en poblaciones. Para ellos hacen una serie de operaciones por cada generación, que son:

1. Selección: Se escoge un número de cromosomas de la población actual aleatoriamente. De esta manera escogemos desordenadamente los padres de la próxima generación. Por cada dos cromosomas hacemos el llamado torneo binario, por el cual seleccionamos el cromosoma con menos coste objetivo.
2. Cruce: Operación por la cual se crea un hijo a partir de dos padres, con una probabilidad Pc. Los padres pertenecen al conjunto previamente seleccionado. Para nuestros algoritmos, por cada padre obtendremos dos hijos. De esta manera mantenemos el número de individuos entre generaciones.
3. Mutación: Operación por la cual se modifica un gen aleatorio de un cromosoma aleatorio del conjunto de hijos con una probabilidad Pm obtenidos en el paso anterior.
4. Reemplazo: En este paso se sustituyen los padres de la generación anterior por los hijos obtenidos, dando paso a la siguiente. Por cada hijo obtenido se sustituye un padre.

Para los algoritmos implementados sólo se mantiene igual la operación de mutación.

Tras estas operaciones, pasaremos a la siguiente generación y repetiremos. Para ambos, el criterio de parada es hacer 100000 evaluaciones de la función objetivo. Al terminar se devolverá el mejor cromosoma de la última generación.

En este experimento las poblaciones de los algoritmos genéticos constan de 50 individuos.

3.3.1. Algoritmo Genético Generacional

La diferencia principal de este algoritmo con respecto el estacionario es que en este caso cada generación de hijos reemplaza completamente a su padre, a excepción del mejor cromosoma que se mantiene.

De esta manera, el operador de selección hace torneos binarios de dos cromosomas seleccionados aleatoriamente hasta conseguir una población prima con el mismo número de individuos.

A continuación, hace la operación de cruce. En un principio sería necesario calcular para cada pareja de padres si estos se cruzan o no bajo una probabilidad Pc. Sin embargo, tras la operación de selección hemos conseguido un orden aleatorio de cromosomas, por lo cual podemos simplemente estimar el número de cruces que serán necesarios (Pc * (numeroCromosomas / 2) y cruzar este número de padres en orden, obteniendo dos hijos por cada cruce. Más adelante comentaré cómo es la operación de cruce, ya que he implementado dos algoritmos de cruce distintos.

Después, hacemos la operación de mutación. Hacemos un procedimiento similar al de cruce (calculamos el número estimado de mutaciones y mutamos los primeros cromosomas de la poblacion). Sin embargo, para cada cromosoma debemos elegir qué gen mutamos. Para ellos elijo un gen *dormido* (valor 0) y un gen *activo* (valor 1) aleatoriamente y los intercambio.

Por último, hacemos el reemplazo. Para ello reemplazamos completamente la población con la población prima obtenida, con una excepción. Al ser una versión elitista, mantenemos el mejor cromosoma de la población anterior, el cual lo usamos para reemplazar el peor de la población prima. Tras esto, comprobamos si es necesario actualizar el mejor cromosoma y volvemos a iterar.

A continuación, muestro el pseudocódigo de este algoritmo:

```
generacional(poblacion, m, datos):

    numCruces <- 0.7 * (numCromosomas/2)
    numMutaciones <- 0.1 * (numCromosomas * m)
    poblacion' <- poblacion
    mejorSolucion <- mejor(poblacion)


    t <- 0

    while (t < 100000) do:

        #Operacion de seleccion
        for(i in numCromosomas) do:
            fila1 = Random(0, numCromosomas - 1)
            fila2 = Random(0, numCromosomas - 1)

            //Torneo binario
            if(coste(fila1) < coste(fila2)):
                poblacion'(i) <- poblacion(fila1)
            else:
                poblacion'(i) <- poblacion(fila2)

        end for

        #Operador de cruce
        for (i in numCruces) do:
            hijo1, hijo2 <- cruza(poblacion'(i), poblacion'(i+1))
            poblacion'(i) <- hijo1
            poblacion'(i+1) <- hijo2
        end for

        #Operador de mutación
        for(i in numMutaciones) do:
            cromosomaAleatorio <- Random(0, numCromosomas -1)

            genesActivos <- elementos (poblacion' == 1)
            genesDormidos <- elementos (poblacion' == 0)

            genActivo <- Random(genesActivos)
            genDormido <- Random(genesDormidos)

            poblacion'(cromosomaAleatorio).intercambia(genActivo, genDormido)
        end for

        #Evaluacion
        for(i in numCromosomas) do:
            coste(i) <- calculaCoste(poblacion'(i))
            t++
        end for

        #Reemplazo
        poblacion = poblacion'

        #Introducir mejor solución de la anterior generacion y actualizar
        #Si la peor solucion de la generacion actual es peor que la mejor de la generacion anterior
        if(coste.maximo > coste(mejorSolucion):
            poblacion.maximo <- mejorSolucion
            mejorSolucion <- poblacion.minimo

    end while

    return poblacion.minimo

nd
```

3.3.2. Algoritmo Genético Estacionario

En este caso, el operador de selección hace torneos binarios de únicamente dos cromosomas seleccionados aleatoriamente, los cuales los obtenemos al conseguir cuatro cromosomas aleatorios y aplicar el torneo binario dos veces. Por tanto, nuestra población prima consta únicamente de dos individuos.

A continuación, hace la operación de cruce. Como sólo tenemos dos individuos, sólo debemos hacer dos cruces. A diferencia del caso anterior, ahora debemos “probar suerte” dos veces, es decir, obtener un número aleatorio y si este se corresponde con la probabilidad de cruce Pc obtendremos un hijo. De esta manera, por cada iteración podemos obtener dos, uno o ningún hijo.

Después, hacemos la operación de mutación. Procedemos de una manera similar al algoritmo generacional, con la diferencia que en lugar de haber un número de mutaciones fijo, hacemos lo mismo que en el cruce pero con probabilidad Pm. Al igual que antes, puede que muten los dos, uno o ninguno.

Por último, hacemos el reemplazo. Seleccionamos los dos peores cromosomas de la población y los reemplazamos por los dos cromosomas obtenidos en el proceso.

A continuación, muestro el pseudocódigo de este algoritmo:

```
estacionario(poblacion, m, datos):

    t <- 0

    while (t < 100000) do:

        #Operacion de seleccion

        #Padre 1
        cromosoma1 <- Random(0, numCromosomas -1)
        cromosoma2 <- Random(0, numCromosomas -1)

        #Torneo binario
        if(coste(cromosoma1) < coste(cromosoma2))
            padre1 <- cromosoma1
        else
            padre1 <- cromosoma2

        #Padre 2
        cromosoma1 <- Random(0, numCromosomas -1)
        cromosoma2 <- Random(0, numCromosomas -1)

        #Torneo binario
        if(coste(cromosoma1) < coste(cromosoma2))
            padre2 <- cromosoma1
        else
            padre2 <- cromosoma2


        #Operador de cruce

        hijo1, hijo2 <- cruza(padre1, padre2)
```

```
        #Operador de mutación

        probabilidadHijo1 <- Random(0,1)

        if(probabilidadHijo1 <= 0.1):

            genesActivos <- elementos (hijo1 == 1)
            genesDormidos <- elementos (hijo1 == 0)

            genActivo <- Random(genesActivos)
            genDormido <- Random(genesDormidos)

            hijo1.intercambia(genActivo, genDormido)

        probabilidadHijo2 <- Random(0,1)

        if(probabilidadHijo2 <= 0.1):

            genesActivos <- elementos (hijo2 == 1)
            genesDormidos <- elementos (hijo2 == 0)

            genActivo <- Random(genesActivos)
            genDormido <- Random(genesDormidos)

            hijo2.intercambia(genActivo, genDormido)


        #Evaluacion
        coste(hijo1) <- calculaCoste(hijo1)
        coste(hijo2) <- calculaCoste(hijo2)
        t <- t+2

        #Reemplazo
        if(coste.maximo > coste(hijo1))
            poblacion.maximo <- hijo1

        if(coste.maximo > coste(hijo2))
            poblacion.maximo <- hijo2

    end while

    return poblacion.minimo

end
```

3.3.3. Algoritmo de cruce uniforme

Este algoritmo de cruce consiste en: para cada gen de un cromosoma elegimos el valor del mismo gen de uno de sus padres elegido aleatoriamente. Es un proceso muy sencillo, sin embargo, la mayoría de las veces resultará en un cromosoma incorrecto ya que no tendrá *m* genes *activos*, por lo cual no sería una respuesta válida para nuestro problema. Por ello, es necesario un proceso de reparación.

En primer lugar comprobamos el número de genes *activos* del hijo. Si es correcto, lo devolvemos. En caso contrario, comprobamos si tiene genes *activos* de más o de menos.

El procedimiento es similar para ambos casos. Si tiene genes de más, buscamos el gen que cuya diferencia de coste (distancia al resto de genes) con la media del coste sea mayor. En el otro caso, el que cuya diferencia sea la menor. En el primer caso borramos ese gen, es decir, lo ponemos a cero y al contrario en el segundo caso.

A continuación, muestro el pseudocódigo:

```
cruceUniforme(cromosoma1, cromosoma2, m, datos):

    #Cruce
    for(i in genes(hijo)) do:
        padreAleatorio = Random(1,2)

        if(padreAleatorio == 1): hijo(i) <- cromosoma1(i)
        if(padreAleatorio == 2): hijo(i) <- cromosoma2(i)
    end for

    #Reparacion

    #Sobran genes
    if(suma(genes(hijo)) > m):
        media <- calculaDistanciaTotal(hijo) / suma(genes(hijo))

        while(suma(genes(hijo)) > m) do:

            genesActivos <- genes(hijo) == 1
            diffMediaMax <- 0

            for(i in genesActivos) do:
                diff <- 0
                for(j in genes(hijo)) do:
                    diff <- diff + abs(datos(j, genesActivos(i)) - media)
                end for

                #Si es el gen mas alejado de la media lo guardamos
                if(diff > diffMediaMax)
                    diffMediaMax <- diff
                    genAEliminar <- genesActivos(i)
            end for

            hijo(genAEliminar) <- 0

            #Actualizamos la media del coste
            media <- calculaDistanciaTotal(hijo) / suma(genes(hijo))
        end while
```

```
    #Faltan genes
    if(suma(genes(hijo)) < m):
        media <- calculaDistanciaTotal(hijo) / suma(genes(hijo))

        while(suma(genes(hijo)) < m) do:

            genesDormidos <- genes(hijo) == 1
            diffMediaMax <- MAX_INT

            for(i in genesDormidos) do:
                diff <- 0
                for(j in genes(hijo)) do:
                    diff <- diff + abs(datos(j, genesDormidos(i)) - media)
                end for

                #Si es el gen mas alejado de la media lo guardamos
                if(diff < diffMediaMax)
                    diffMediaMax <- diff
                    genAñadir <- genesDormidos(i)
            end for

            hijo(genAñadir) <- 1

            #Actualizamos la media del coste
            media <- calculaDistanciaTotal(hijo) / suma(genes(hijo))
        end while

    return hijo
end
```

3.3.4. Algoritmo de cruce de posición

Este algoritmo de cruce es más sencillo que el anterior puesto que no necesita un procedimiento de reparación.

El primer paso es seleccionar los genes cuyo valor coincidan en ambos padres y asignarlos a un hijo temporal

Después, creamos dos descendientes auxiliares con los valores del primer padre que NO coinciden con el segundo para después mezclarlos aleatoriamente.

Por último, asignamos cada uno de estos descendiente auxiliares a dos hijos y los devolvemos.

A diferencia del anterior, en este caso he optado por crear una función que devuelva directamente los dos hijos ya que me parecía más eficiente al tener que recorrer los valores del padre una única vez.

A continuación, muestro el pseudocódigo:

```
crucePosicion(cromosoma1, cromosoma2, m, datos):

    #Asigno los elementos comunes entre ambos
    hijo <- elementos(cromosoma1 == cromosoma2)

    #Asignacion aleatoria del resto del cromosoma1
    noComunes <- elementos(cromosoma1 != cromosoma2)
    padre_descendiente1 <- cromosoma1(noComunes)
    padre_descendiente2 <- cromosoma1(noComunes)


    #Desordeno estos descendientes, que son los resto del padre

    for(i in cromosoma1) do:
        aux <- Random(0, cromosoma1.size() -1)
        padre_descendiente1.intercambia(i, aux)

        aux <- Random(0, cromosoma1.size() -1)
        padre_descendiente2.intercambia(i, aux)

    end for

    hijo1 <- hijo
    hijo2 <- hijo

    elementos(hijo1(noComunes)) <- padre_descendiente1
    elementos(hijo2(noComunes)) <- padre_descendiente2

    return hijo1, hijo2
end
```

3.5. Algoritmo Memético

Este algoritmo es muy similar a los genéticos previamente distintos. La diferencia con estos es que aplicamos un procedimiento de optimización local cada cierto número de generaciones a nuestra población.

Para mi implementación he optado por usar el AGG-Uniforme, ya que de los algoritmos generacionales es el que mejores resultados me ha dado.

Por lo tanto, este algoritmo es igual que el AGG-Uniforme, salvo que hace una operación extra de optimización usando la búsqueda local también implementada en el proyecto. Esta operación se sitúa después de la mutación y antes de la evaluación.

Hay tres versiones de esta implementación, las cuales se diferencian en el número de veces que se llama a BLM y para qué cromosomas:

- AM(10, 1.0): Cada 10 generaciones aplica la búsqueda local en cada cromosoma de la población.
- AM(10, 0.1): Cada 10 generaciones aplica la búsqueda local con una probabilidad del 10% a cada individuo.
- AM(10, 0.1mej): Cada 10 generaciones aplica la búsqueda local a un subconjunto del 10% de lo individuos, los cuales son los mejores de la población.

En este experimento, las poblaciones del algoritmo memético constan de 10 individuos, lo cual hace que en el último caso solo se aplique la BLM en el mejor cromosoma.

El pseudocódigo de la BLM no ha cambiado con el respecto al de la práctica uno, en cambio, su implementación ha cambiado notablemente debido al cambio de Eigen a Armadillo, además de un par de pequeños errores.

3.6. Algoritmo de Enfriamiento Simulado (ES).

Al igual que el algoritmo de Búsqueda Local, el algoritmo de Enfriamiento Simulado trata en encontrar la mejor solución dentro de un vecindario. Sin embargo, este cuenta con un criterio probabilístico de aceptación de soluciones basado en termodinámica. Es decir, que no solo se aceptan las soluciones con un mejor coste sino que hay un procedimiento probabilístico que también se tiene en cuenta a la hora de aceptarlas.

Su objetivo es el de evitar que la búsqueda se atasque en óptimos locales, por lo cual lo que hacemos es permitir que algunos movimientos sean hacia soluciones peores, pero debemos hacerlo de manera controlada. El objetivo es hacer que la frecuencia de escape sea mayor al principio de la ejecución y vaya disminuyendo según ésta avanza.

Para ello usamos una variable llamada **Temperatura T**, cuyo valor determina en qué medida pueden ser aceptadas soluciones vecinas peores.

Para esta práctica se ha usado el mecanismo de enfriamiento del *Esquema de Cauchy modificado*, el cual determina cuanto disminuye la temperatura cada vez que se generan L(T) soluciones vecinas y también determina el número de iteraciones totales que tendrá el algoritmo.

El criterio de parada implementado es cuando T < Tf (fijada a un valor muy pequeño), cuando se superan las 100000 iteraciones o cuando se genera todo L(T) sin hacer ningún intercambio, mientras no se cumplan las condiciones continúa la búsqueda.

A continuación muestro el pseudocódigo:

```
#Algoritmo de Enfriamiento Simulado
ES(m, datos):
  max_vecinos <- 10 * m
  max_exitos <- 0.1 * max_vecinos
  Tf <- 10^(-3)

  while(Tf > T) do:
    sel <- sel_aleatoria()
    T <- (0.3 * coste(sel)) / -log(0.3)
  end while

  beta <- T - Tf / (100000/max_vecinos) * T * Tf

  #Bucle principal
  while(Tf < T and contador < 100000 and exitos != 0) do:
    seHaceUnIntercambio <- false
    exitos <- 0

    for (i in max_vecinos and exitos < max_exitos) do:
      contador <- contador + 1

      #interA debe ser un valor en sel, interB un valor fuera de sel
      while (not inSel(interA) or not outSel(interB)) do:
        interA <- Random(datos)
        interB <- Random(datos)
      end

      seIntercambia <- Int(sel, T*numEnfriamientos, interA, interB, datos)

      if (seIntercambia) do
```

```
        exitos <- exitos + 1

        if (coste(sel) < coste(mejor_sel) do
          mejor_sel <- sel
        end
    end for

    numEnfriamientos <- numEnfriamientos + 1
    T <- T / 1 + (beta*T)

  end while
  return mejor_sel
end
```

Para este algoritmo y para ILS se ha usado un algoritmo de intercambio *Int* levemente modificado para incluir la condición probabilística de aceptación. Muestro el pseudocódigo a continuación:

```
#Operador de intercambio para ES
Int (sel, delta, interA, interB, datos):
  sel_aux <- sel
  seIntercambia <- false

  for(i in sel) do:
    if(i != interA)          distancia_v <- distancia_v + datos(interB, sel(i))
    coste(sel_aux(i)) <- coste(sel_aux(i)) - datos(sel(i), interA)
    coste(sel_aux(i)) <- coste(sel_aux(i)) + datos(sel(i), interB)
  end for

  diferencia_coste <- max(distancia_v, max(sel_aux)) - min (distancia_v, min(sel_aux)) - coste(sel)
  pasaProbAceptacion <- Random(0,1) <= exp((-diferencia_coste) / delta)

  if(diferencia_coste < 0 or pasaProbaceptacion):
    seIntercambia <- true
    sel <- sel_aux
    intercambia(sel, interA, interB)
  end

  return seIntercambia
end
```

3.7. Algoritmo de búsqueda multiarranque básica (BMB)

El funcionamiento de este algoritmo es bien sencillo. El objetivo es lanzar muchas veces una búsqueda local, partiendo de una solución aleatoria distinta cada vez, y quedarse con la mejor solución de todas.

A continuación muestro el pseudocódigo:

```
#Busqueda multiarranque básica

BMB(m, datos):

  s0 <- generaSelAleatoria()
  mejor_sel <- s0

  for(i in 10) do:
    s <- BusquedaLocal(s0)

    if(coste(s) < coste(mejor_sel)):
      mejor_sel <- s
    end

    s0 <- genereaSelAleatoria
  end for

  return mejor_sel
end
```

3.8. Algoritmos de búsqueda local reiterativos basados en óptimos (ILS).

Los ILS están basados en la aplicación repetida de un algoritmo de búsqueda local a una solución inicial que se obtiene por la mutación de un óptimo local previamente encontrado.

Los pasos son (para esta práctica en concreto): generar una solución inicial, mutar la mejor solución, aplicar búsqueda local al resultado y aplicar un criterio de aceptación basado en el mejor coste.

Para la mutación, cada vez que se muta aplicamos el operador de intercambio sobre t=0.3 * m elementos de la solución para provocar un cambio brusco.

En esta práctica se han implementado dos versiones del algoritmo ILS, una usando la búsqueda local de la práctica 1 y otra usando la búsqueda por enfriamiento simulado implementada en la práctica 3. Para ambas el código del algoritmo es el mismo.

A continuación muestro el pseudocódigo.

```
#Búsqueda Local Reiterada

ILS(m, datos):
  s0 <- generaSelAleatoria()
  mejor_sel <- BusquedaLocal(s0)
```

```
  for(i in 10) do:
    sprima <- mutacion(mejor_sel)

    sprima2 <- BusquedaLocal(sprima)  # or BusquedaES(sprima)

    if (coste(sprima2) < coste(mejor_sel):
      mejor_sel <- sprima2
    end
  end for

  return mejor_sel
end
```

4. Procedimiento para desarrollar la practica.

El proyecto se ha desarrollado completamente en C++. Este consta de varios archivos (cabecera y .cpp), siendo estos *lectorDatos, solucionGreedy, solucionBLM* y por último *main*. Además, incluye la interfaz proporcionada por los profesores para la generación de números aleatorios *random.hpp*

- *main*: Desde este archivo se generan los datos de salida necesarios para la práctica. Define las semillas para los números aleatorios, llama a los algoritmos Greedy, BLM, las cuatro versiones de algoritmos genéticos y las 3 versiones de algoritmos meméticos, pasándole los datos de cada fichero. Los algoritmos Greedy y BLM son llamados 5 veces cada uno con una semilla distinta y luego se calcula la media de sus resultados. Finalmente, calcula el tiempo invertido en cada llamada y guarda la media de tiempo y la media de coste por archivo de datos en nuevos archivos en formato .cvs para su posterior análisis en una hoja de cálculo.
- *lectorDatos*: Se encarga de abrir los ficheros de datos, crear y almacenar de forma correcta las matrices de datos para los experimentos. Está diseñado de tal manera que solo es necesario pasarle el número del archivo como parámetro. De esta manera podemos iterar cómodamente ya sea desde código C++ o desde un script *bash.*
- *solucionGreedy*: Contiene el algoritmo principal y su función heurística
- *solucionBLM*: Contiene el algoritmo principal, la función de intercambio y la función de mezcla.
- *solucionGenetica:*Consiste en una clase *Genetico,* la cual contiene los métodos que implementan los algoritmos *AGG* y *AGE*, los operadores de cruce *Uniforme* y *Posición*, además de las variables necesarias para estos métodos y algunos métodos auxiliares.
- *solucionMemetica:* Consiste en una clase *Memetico,* la cual contiene los métodos que implementan los algoritmos AGG y BLM, el operador de cruce *Uniforme* y el operador de intercambio *Int*, además de las variables necesarias para estos métodos y algunos métodos auxiliares.

- *solucionES:* Contiene las funciones para el algoritmo principal, la búsqueda local y otras funciones auxiliares.
- *solucionBMB:* Contiene las funciones para el algoritmo principal, la búsqueda local y otras funciones auxiliares.
- *solucionILS:* Consiste en una clase *ILS*, la cual contiene los métodos que implementan el algoritmo principal, la búsqueda local, el operador de mutación y otras funciones auxiliares.
- *solucionILS-ES*: Consiste en una clase *ILSES*, la cual contiene los métodos que implementan el algoritmo principal, la búsqueda local, el operador de mutación y otras funciones auxiliares.

Para la implementación de los conjuntos de datos, tanto de los de entrada como la representación de las soluciones he optado por usar conjuntos de la STL de C++ (como *vector* y *pair*) y conjuntos de la librería Armadillo.

En la anterior entrega decidí usar la librería Eigen, pero tras comprobar que los métodos necesarios para modificar el tamaño de los conjuntos, como eliminar o añadir elementos, no funcionaban decidí probar Armadillo. Una vez actualicé a la nueva librería e hice algunos ajustes los algoritmos Greedy y BLM dejaron de dar soluciones erroneas. Esta librería provee de un manejo de conjuntos muy cómodo e intuitivo, además de una gestión dinámica de estos realmente eficaz y sencilla.

4.1. Manual de uso.

El proyecto hace uso de CMake para la creación del Makefile y de la compilación. Se compone de tres carpetas: src, build y datos. *src* contiene todo el código implementado, *datos* todos los datos de entrada y los de salida, incluyendo los mejores resultados que he obtenido y *build* contiene los archivos de CMake, el Makefile. El archivo ejecutable *MH* se encuentra en la carpeta *bin*, junto con un README.txt con la explicación de los ficheros antes mostrada.

El uso es tan sencillo como ejecutar el archivo *MH*. Si se desea recompilar solo hay que hacer uso del Makefile. Una vez ejecutado el programa muestra una interfaz de texto simple que deja decidir si se quiere ejecutar un algoritmo una única vez, donde dejará especificar el caso y la semilla a utilizar, o si desea ejecutar el experimento completo (50 archivos de datos con 5 semillas cada uno para los dos algoritmos).

Importante. Como ya he mencionado antes, este proyecto hace uso de la biblioteca **Armadillo**, por lo que si se desea volver a compilar será necesario tenerla instalada.

5. Experimentos y análisis de resultados.

Como he explicado en apartados anteriores, el experimento consiste en probar los algoritmos Greedy, BLM, AGG-Uniforme, AGG-Posicion, AGE-Uniforme, AGE-Posicion, AM(10, 1.0), AM(10, 0.1) y AM(10, 0.1mej) para los 50 casos GDK-b de la MDPLIB. Para los algoritmos Greedy y BLM se ejecutan 5 veces por cada caso con las semillas {23, 32, 46, 69, 72}, después se calcula la media de los costes obtenidos y el tiempo de ejecución.

Para obtener los mejores datos posibles dentro de mis posibilidades el proyecto ha sido ejecutado en mi máquina en un entorno de sólo terminal por lo que eran muy pocos los procesos compitiendo por el uso de CPU (esto sólo afecta a los tiempos de ejecución ya que los resultados vienen dados por la semilla generadora de números aleatorios).

A continuación muestro una tabla con los resultados globales de todos los algoritmos implementados:

RESULTADOS GLOBALES		
Algoritmo	**Desviación**	**Tiempo**
Greedy	77,59	0,9
BLM	78,29	35,97
AGG-Uniforme	66,92	5748,18
AGG-Posicion	67,89	3322,3
AGE-Uniforme	62,83	1632,48
AGE-Posicion	70,64	1097,26
AM(10, 1.0)	73,29	783,26
AM(10, 0.1)	68,08	927,3
AM(10, 0.1mej)	67,41	970
ES	72,28	23,06
BMB	76,62	14,22
ILS	76,48	14,02
ILS-ES	65,26	204,38

5.1. Análisis de BLM y Greed (práctica 1).

Los resultados medios de ambos algoritmos en la mejor ejecución son:

ALGORITMO GREEDY	
Media Desv:	83,74
Media Tiempo(ms):	18,876

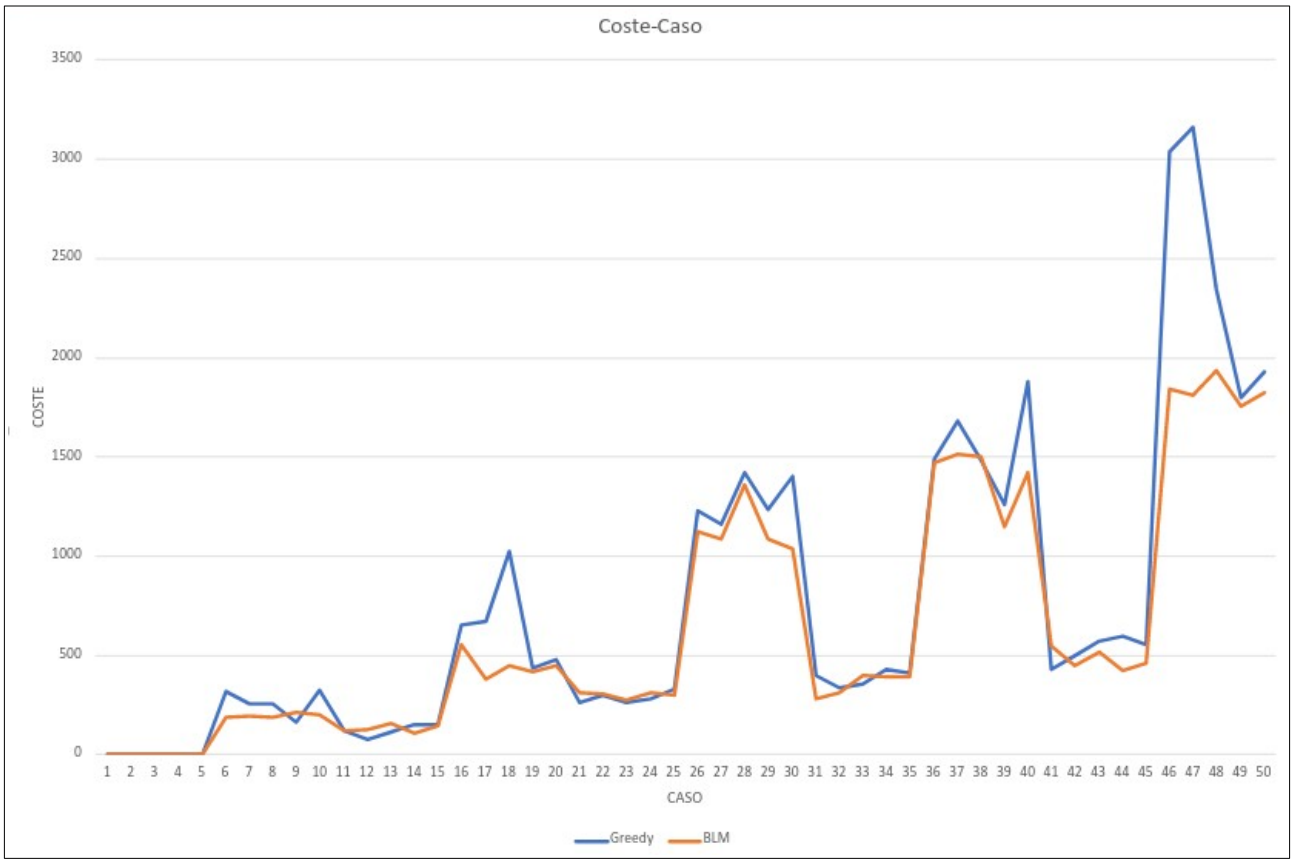
ALGORITMO BLM	
Media Desv:	82,72
Media Tiempo(ms):	85,924

Lo primero que observamos es que ambos algoritmos tienen un rendimiento muy similar en cuanto al coste. La media de la desviación típica en cuanto al mejor resultado posible muestra cómo ambos algoritmos consiguen un coste realmente parecido. Esto nos hace pensar es indiferente cual escoger para este problema.

Por otra parte, resalta la gran diferencia en tiempos de ejecución. El tiempo medio consumido por caso es del orden de 4 veces mayor para el algoritmo BLM. Esto cobra gran importancia en este experimento en concreto ya que es el dato diferencial más significativo y nos conduce a decidirnos por el algoritmo Greedy para mejor candidato entre los dos.

Sin embargo, echando un rápido vistazo a los resultados obtenidos observé cómo, de media, los costes obtenidos en el algoritmo BLM eran menores que el algoritmo Greedy, lo cual tiene sentido ya que hace un análisis más profundo de los datos y prueba muchas más opciones.

Decidí por hacer una comparación gráfica para comparar más fácilmente este fenómeno, obteniendo la siguiente gráfica de líneas:



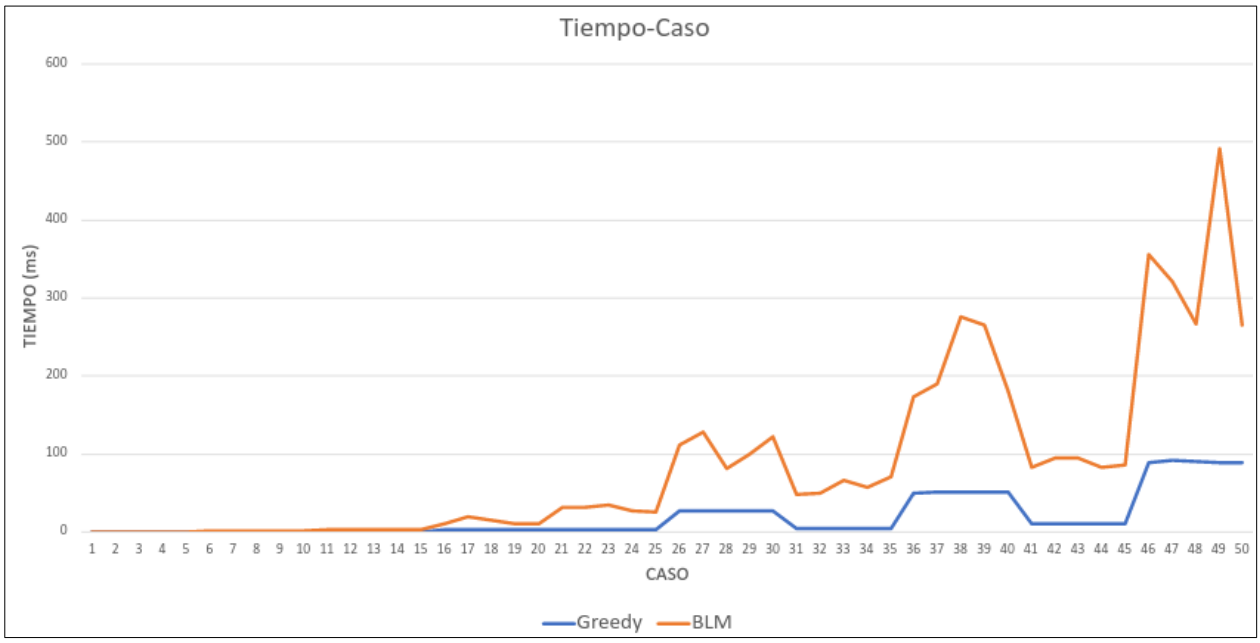
Se aprecia claramente cómo los costes obtenidos por el algoritmo BLM son (casi) siempre menores que en el caso Greedy, llegando a haber diferencias bastante significativas en algunos grupos de casos.

Ambos algoritmos siguen la misma tendencia, pero observamos cómo en los conjuntos de casos [16,20] y [46,50] las diferencias son significativas. Estos casos se caracterizan por tener un tamaño del conjunto de selección *m* más grande con respecto al número total de localizaciones. El ratio para ambos es 10/3, siendo el primer intervalo 50/15 y el segundo 150/45.

Esto se debe a la función de búsqueda de BLM. Mientras que Greedy debe explorar el espacio de búsqueda casi al completo *m* veces, BLM no tiene esa limitación, ya que seguirá explorando hasta hacer una pasada por un vecindario al completo sin encontrar una mejor solución (en nuestro caso está limitado a 100000, lo cual puede ser el motivo de que el resto de costes sean tan similares). De esta manera, BLM tiene muchas más oportunidades de encontrar soluciones con mejor coste.

Aunque proporciones mejores costes, la mejora no es muy significativa y si nos fijamos en los tiempos de ejecución nos replanteamos si realmente es BLM mejor que Greedy para este experimento en concreto.

A continuación, muestro una comparación de tiempos de ejecución con respecto a los casos estudiados:



Vemos cómo el tiempo de ejecución de BLM se dispara para los casos con un conjunto de datos de mayor dimensión. Mientras que Greedy aumenta ligeramente según hay más datos, nunca llegando a superar los 100 ms, BLM roza los 400 ms en los peores casos.

Esto se debe a la cualidad de BLM que comentábamos antes. Este es capaz de explorar todos los datos muchas más veces que Greedy, lo cual normalmente supone la obtención de mejores resultados.

Sin embargo, como dije al principio de este análisis, las diferencias de coste no son significativas por lo que podríamos elegir un algoritmo u otro indiferentemente (para este grupo de casos). Ambos han demostrado no ser muy buenos para encontrar la mejor solución, eso está claro sólo con mirar la desviación típica, pero el vencedor entre ambos es claro.

Mi conclusión tras este experimente es que, aunque en un principio no lo esperaba, el vencedor entre ambos algoritmos es Greedy, ya que, aunque los resultados no son los mejores ni cercanos a la mejor solución, sus tiempos de ejecución son realmente bajos, no llegando a superar los 100ms en los peores casos y obteniendo resultados de prácticamente 0 segundos para la mitad de ellos.

5.2. Análisis de algoritmos basados en poblaciones (práctica 2)

5.2.1. Análisis de algoritmos genéticos

Los resultados de las cuatro versiones son:

ALGORITMO AGG-U	
Media Desv:	66,92
Media Tiempo(ms):	5748,18
Media Coste	280,002

ALGORITMO AGE-U	
Media Desv:	62,83
Media Tiempo(ms):	1632,48
Media Coste	195,895

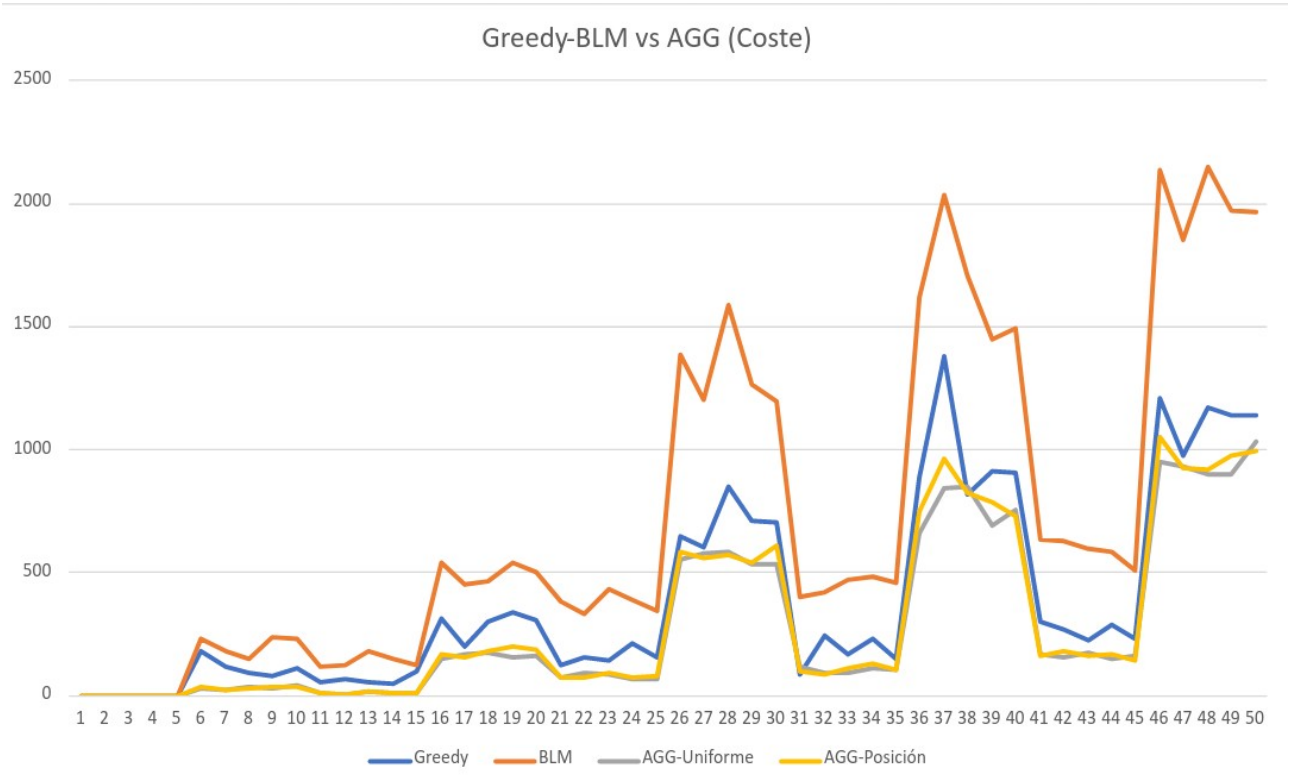
ALGORITMO AGG-P	
Media Desv:	67,89
Media Tiempo(ms):	3322,3
Media Coste	292,259

ALGORITMO AGE-P	
Media Desv:	70,64
Media Tiempo(ms):	1097,26
Media Coste	285,545

Viendo estos datos comprobamos cómo estos algoritmos tienen un rendimiento muy similar en cuanto el coste, con la excepción del Algoritmo Genético Estacionario con cruce Uniforme. Este destaca con diferencia del resto, ya que no solo tiene un coste muy inferior al resto sino que también tiene una media de tiempo bastante buena, siendo la segunda mejor de los cuatro.

Estos datos arrojan bastante luz sobre cómo va a ir el resto del análisis. De todas formas, mostraré una serie de gráficas para comparar estos algoritmos tanto entre sí como con los algoritmos Greedy y BLM.

En primer lugar tenemos una comparación de los algoritmos generacionales con Greedy y BLM:

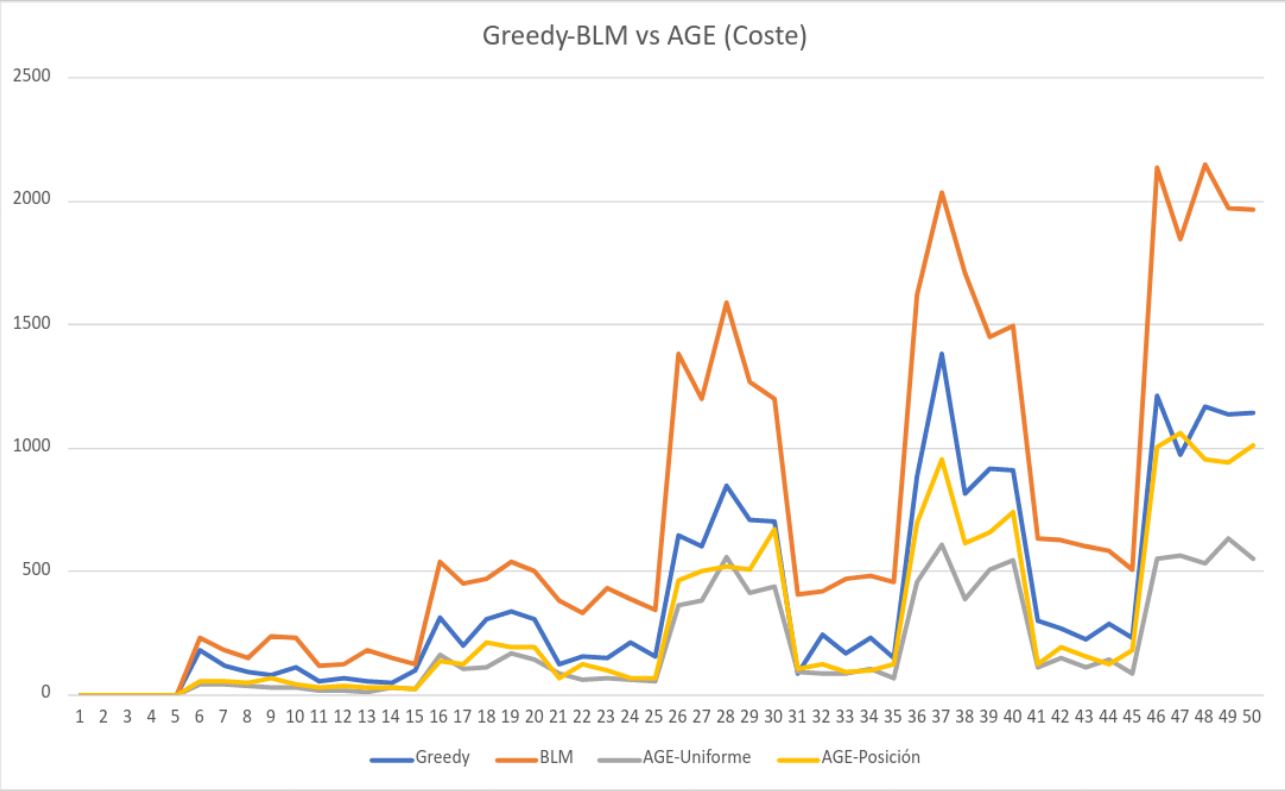

Greedy-BLM vs AGG (Coste)
2500
2000
1500
1000
500
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
Greedy
BLM
AGG-Uniforme
AGG-Posición

Al igual que en el análisis de la práctica anterior, BLM es el que peor resultados devuelve, llegando a picos de más de 2000 de coste.

Aunque la diferencia con Greedy no parezca muy notable, es interesante ver cómo las dos implementaciones del algoritmo genético mantienen cierta estabilidad, no tienen unos picos de coste muy grandes como tiene Greedy. De media obtienen unos datos mucho mejores que los de Greedy.

Por otro lado, la diferencia entre ambas implementaciones, una con cruce uniforme y otra con cruce de posición, no es muy grande. Sin embargo, vemos cómo para los conjuntos de datos que tienen un *m* mayor la implementación con cruce uniforme tiende a quedarse por debajo. Aunque en mi opinión, la mejoría del coste frente a la diferencia del tiempo de ejecución no merece la pena. Si tuviera que elegir un ganador me decantaría por la implementación con cruce de posición.

Pasando a los algoritmos estacionarios, obtenemos la siguiente gráfica:



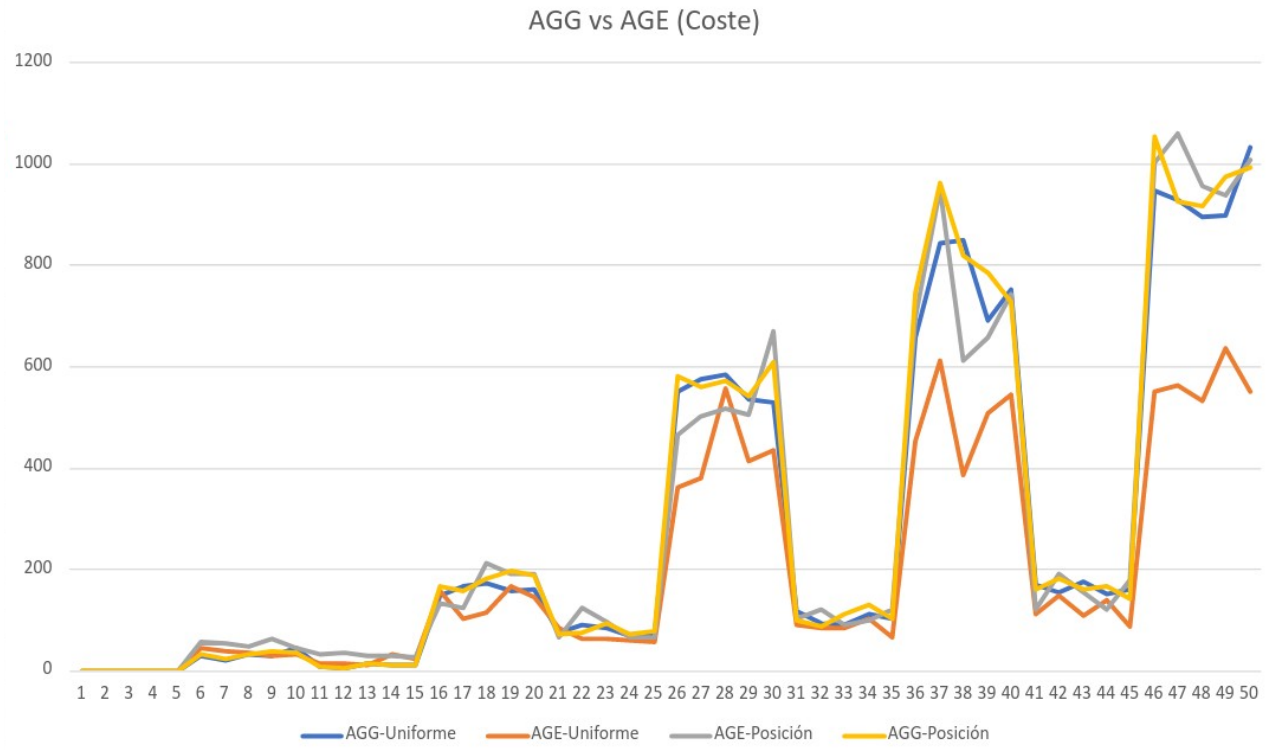
La tendencia es similar al anterior, aunque ahora vemos una diferencia aún mayor con Greedy.

En este caso (como ya veíamos venir desde un principio) la implementación del AGE-Uniforme destaca notablemente, ya que en sus “peores” ejecuciones alcanza picos de poco más de 500, lo cual en comparación de los más de 2000 de BLM o 1000 de Greedy es increíble.

Los algoritmos estacionarios no sólo aportan mejores costes, sino que al tener una implementación con muchos menos bucles (y por tanto iteraciones totales sobre los datos) obtienen unos tiempos de ejecución mucho mejores que los algoritmos generacionales. El hecho de asegurarse que mejoras dos cromosomas en cada iteración aumenta el éxito frente al elitismo de los generacionales.

En adición, el uso del algoritmo de cruce arroja mejores resultados. Esto puede deberse a la operación de reparación, que siempre busca eliminar los genes que aumenten el coste o añadir aquellos que aporten lo mínimo posible.

Por último, compararé los AGG y los AGE directamente:



En esta gráfica queda constancia de lo dicho en el primer párrafo. La mayoría de implementaciones siguen una tendencia similiar mientras AGE-Uniforme destaca, sobre todo para el último grupo de datos.

Sin embargo, viendo esta gráfica me parece interesante mencionar cómo AGE-Uniforme se mantiene en un rango de valores bastante regular para los últimos tres grupos de datos "difíciles". En el primero de ellos tiene su peor ejecución para el archivo 28, mientras que el resto de algoritmos se mantiene más o menos en la misma línea para ese archivo.

5.2.2. Análisis de los Algoritmos Meméticos

Los resultados de los tres experimentos son:

Algoritmo AM(10, 1.0)	
Media Desv:	73,29
Media Tiempo(ms):	783,26
Media Coste	378,693

Algoritmo AM(10, 0.1)	
Media Desv:	68,08
Media Tiempo(ms):	927,3
Media Coste	325,791

Algoritmo AM(10, 0.1mej)	
Media Desv:	67,41
Media Tiempo(ms):	970
Media Coste	318,407

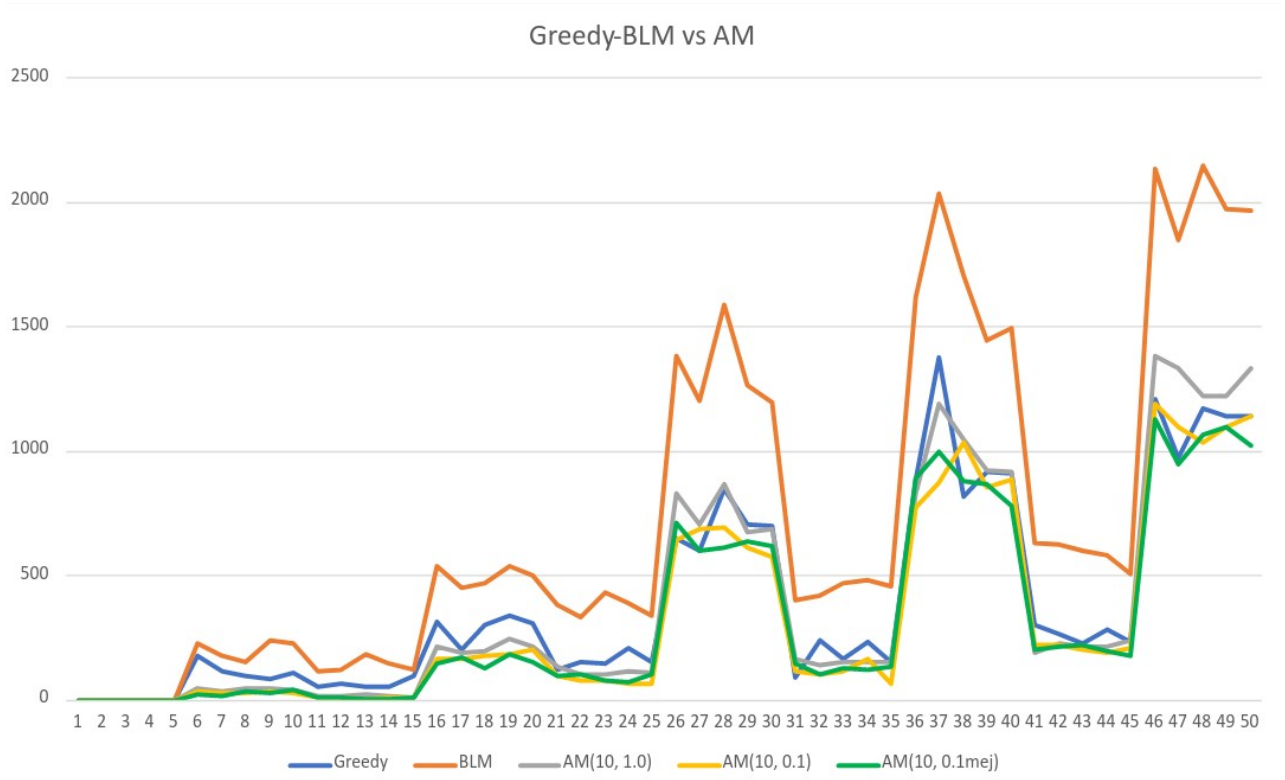
Lo primero que observo es los costes obtenidos son peores que los obtenidos con los AG. Esto puede ser debido a la reducción del tamaño de la población. Al tener un número menor de individuos es más complicado obtener mejores datos.

Sin embargo, cabe destacar que pese a esta reducción los resultados obtenidos no son tan malos como podrían ser. La optimización de los datos usando la BLM hace que los datos obtenidos sean realmente buenos.

Además, los tiempos de ejecución son mucho menores que en los algoritmos genéticos, lo cual se debe obviamente a la reducción del espacio de búsqueda, aunque el número de evaluaciones totales sea el mismo.

Paso a mostrar una serie de gŕaficas para poner estos resultados en perspectiva.

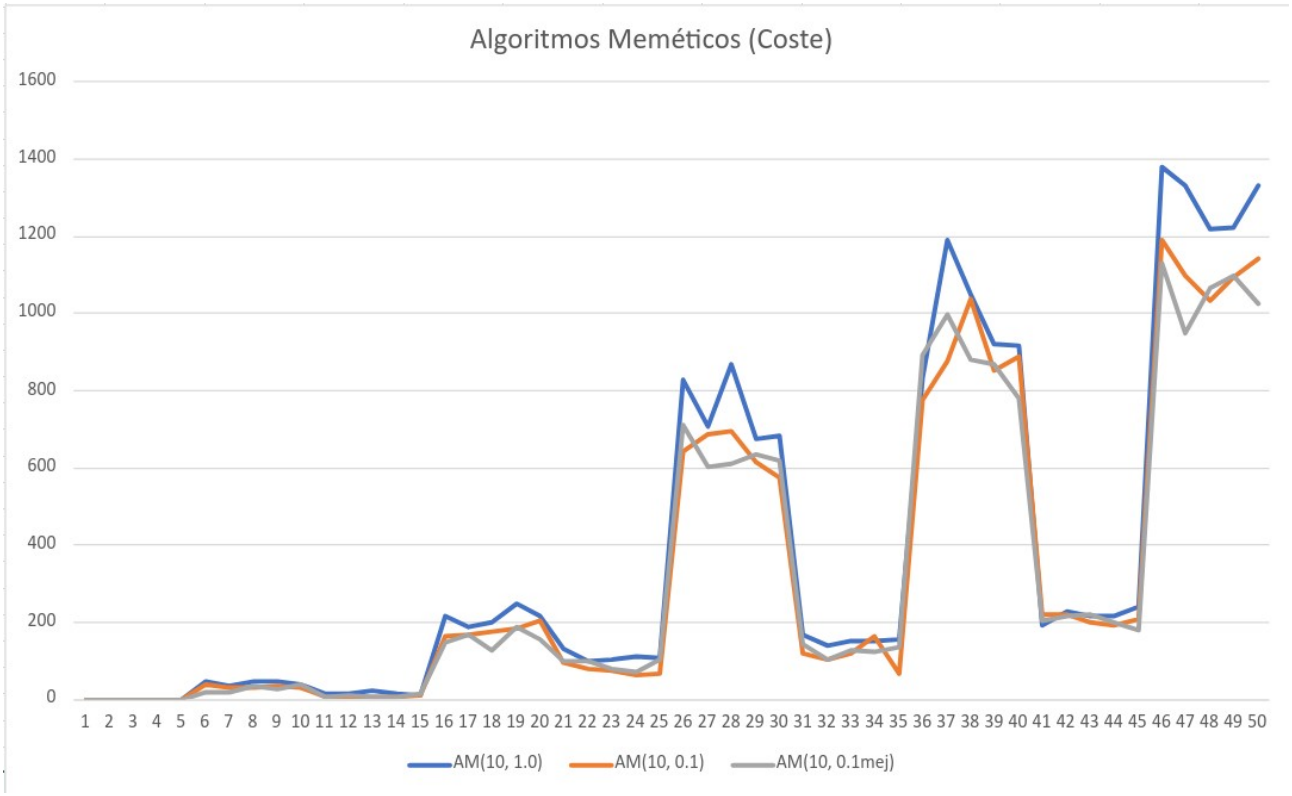
En primer lugar, compararé los AM con Greedy y BLM:



Podemos observar cómo los resultados no distan mucho de los obtenidos con Greedy. En general se obtienen mejores resultados, aunque en el último grupo de datos Greedy supera a dos de las implementaciones y se mantiene más o menos igual que la última. Sin embargo, la tendencia general es obtener mejores resultados, sobre todo en los grupos de datos “fáciles”, donde la diferencia con Greedy es mayor.

Por otra parte están los tiempos de ejecución. Los tiempo de Greedy son rídiculos, creo que si tenemos en cuenta la relación coste-tiempo, Greedy es una opción mucho mejor.

A continuación muestro una gráfica sólo con los algoritmos meméticos:



Vemos cómo usar la búsqueda local en todos los cromosomas de la población no es nada productivo, ya que no sólo aumentan los tiempos de ejecución sino que los resultados son peores.

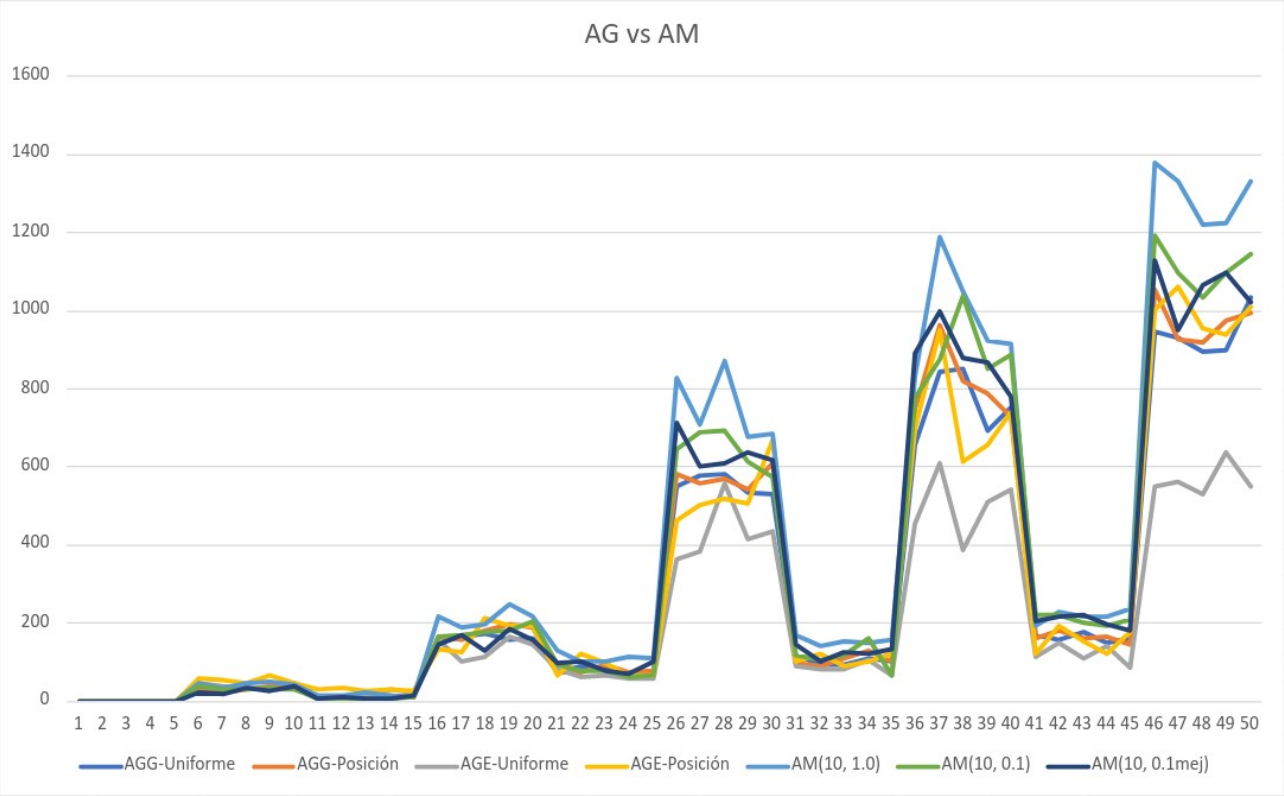
Ahora la duda está entre AM(10, 0.1) y AM(10, 0.1mej). Los resultados son similares, aunque AM(10, 0.1mej) tiende a tener mejores resultados, aunque solo ligeramente. Los tiempos de ejecución tampoco esclarecen mucho las dudas, ya que AM(10, 0.1) obtiene mejores tiempo, creo que debido a que al hacer más llamadas a BLM completa el número de evaluacions total antes.

En mi opinión, mi implementación del algoritmo BLM deja mucho que desear y esto hace que los resultados no sean tan buenos como podrían ser. No es normal que haya tanta diferencia entre Greedy y BLM.

5.2.3. Algoritmos Genéticos vs Meméticos.

La cuestión más importante de esta práctica es qué grupo de algoritmos es mejor. Tras el análisis hecho queda claro que los algoritmos genéticos son mejores con bastante diferencia, mejores incluso que Greedy.

A continuación muestro una gráfica comparando todos los algoritmos implementados:



Sin embargo, al ver la gráfica observamos cómo, a diferencia de AGE-Uniforme, todos se mantienen en un rango no muy amplio de costes. Si bien es cierto que los genéticos dominan en la tendencia general, hay muchos casos donde los meméticos ganan, sobre todo en los primeros grupos de datos.

Al final del todo, el vencedor es el mismo que vaticiné al principio. La combinación de los buenos tiempos de ejecución con los mejores resultados hacen a AGE-Uniforme la mejor opción de todo el proyecto para resolver el problema de la mínima dispersión.

5.3. Análisis de algoritmos basados en trayectorias (práctica 3).

RESULTADOS GLOBALES		
Algoritmo	**Desviación**	**Tiempo**
Greedy	77,59	0,9
BLM	78,29	35,97
ES	72,28	23,06
BMB	76,62	14,22
ILS	76,48	14,02
ILS-ES	65,26	204,38

5.3.1. Comparación de algoritmos de búsqueda local

En esta sección voy a comparar los algoritmos *Greedy, BL* y *ES*. Los resultados obtenidos para todos ellos son:

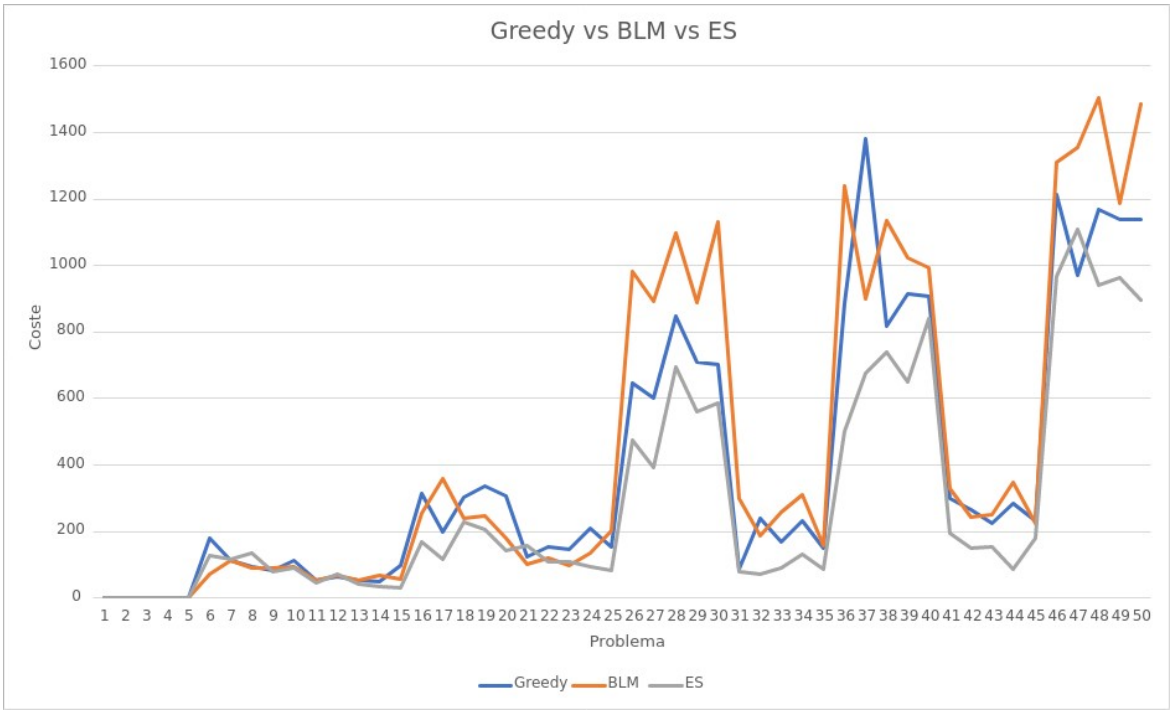
ALGORITMO GREEDY	
Media Desv:	**77,59**
Media Tiempo(ms):	**0,9**
Media Coste	**387,72593**

ALGORITMO BLM	
Media Desv:	**78,29**
Media Tiempo(ms):	**35,972**
Media Coste	**448,993354**

Algoritmo ES	
Media Desv:	**72,28**
Media Tiempo(ms):	**23,06**
Media Coste	**288,336**

A primera vista observamos cómo el algoritmo ES obtiene unos resultados en cuanto a coste mucho mejores de los que obtienen los otros dos y un coste de tiempo bastante mejor que BLM.

Para hacer un análisis en más profundidad añado una tabla que los compara problema a problema a continuación:



Podemos ver cómo para los primeros problemas la diferencia entre ES y los otros dos algoritmos no es muy notable. Sin embargo, para los problemas más difíciles la diferencia entre ellos se hace notar. Cuando se trata de problemas cuyo tamaño de solución (m) es mayor ES rinde mucho mejor que BLM y Greedy.

Esto se debe a la capacidad de ES de escapar de óptimos locales, por ello donde el espacio de búsqueda es más complejo y pueden encontrarse muchos de ellos, BLM suele dar peores resultados ya que devuelve el primer óptimo local que encuentre mientras que ES es capaz de encontrar soluciones mucho mejores.

Todo esto se suma a que sus tiempos de ejecución son bastante menores que los de BLM, por lo que no solo da mejores resultados sino que es más eficiente.

5.3.2. Comparación de algoritmos de trayectorias múltiples.

Los resultados obtenidos en los algoritmos BMB, ILS e ILS-ES son los siguientes:

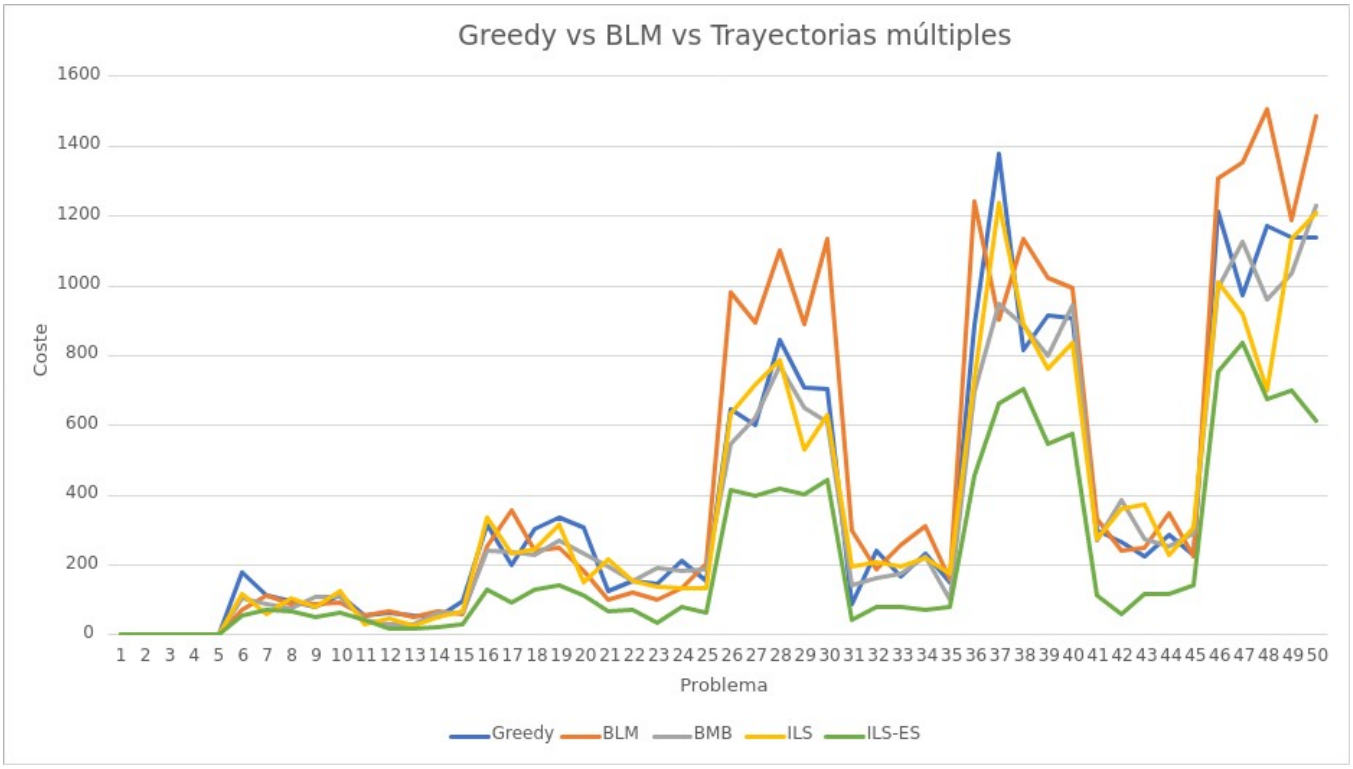
Algoritmo BMB	
Media Desv:	**76,62**
Media	**14,22**
Media Coste	**358,780**

Algoritmo ILS	
Media Desv:	**76,48**
Media	**14,02**
Media Coste	**360,713**

Algoritmo ILS-ES	
Media Desv:	**65,26**
Media	**204,38**
Media Coste	**217,283**

Lo primero que se aprecia es que los tres son mejores que BLM en todos los aspectos. No solo eso, sino que la implementación de ILS que usa ES en lugar de BLM es mucho mejor, de hecho, fijándonos en estos datos es el mejor algoritmo de toda la práctica para resolver el problema MDD, a excepción de AGE-Uniforme.

Como en casos anteriores, a continuación incluyo un gráfica comparando estos tres algoritmos con BLM y Greedy:



Si bien es cierto que de media estos algoritmos son mejores que BLM podemos ver ciertas diferencias entre ellos.

En primer lugar, observo que los resultados de ILS tienden a acercarse a los de BLM para algunos casos, principalmente en el grupo de problemas [35,40), donde incluso BLM le supera en un caso. Esto me resulta extraño ya que la tendencia del resto de algoritmos es que el coste aumenta en ese mismo caso, solo BLM consigue mejorar el coste con respecto al anterior. Esto puede deberse a que justo para la semilla elegida BLM alcanza un muy buen óptimo local, mejor que el que alcanzan el resto de adversarios. Pero en definitiva, vemos como para todos los casos ILS es mejor que Greedy y BLM al hacer varias búsquedas en lugar de una, pero peor que ILS.

Con BMB se obtienen unos resultados bastante buenos, ya que aunque estos no sean los mejores, los obtiene en muy poco tiempo. También cabe destacar que los costes obtenidos son muy similares a los obtenidos a ILS. Es curioso, ya que la única diferencia es que el segundo implementa un operador de mutación que el primero no tiene. Esta diferencia debería ser crucial y darnos resultados muy distintos pero, en mi opinión, al ser el criterio de aceptación igual en ambos la probabilidad de que se encuentre una buena solución en las primeras iteraciones hace que el operador de mutación pierda importancia y por tanto ambos algoritmos pueden llegar incluso a las mismas soluciones. Con los resultados obtenidos no sabría cual de los dos escoger, quizá BMB ya que su implementación es mucho más sencilla.

Por último, vemos como ILS-ES consigue resultados mucho mejores para todos y cada uno de los problemas. Esto va en consonancia con lo analizado en el punto anterior, donde veíamos que el algoritmo ES daba mucho mejores resultados que BLM. No es de extrañar por tanto que al implementarlo en ILS los resultados obtenidos sean mejores que al usar BLM. Sin embargo, la diferencia de resultados es más que notable, ya que ILS pasa de ser el segundo mejor algoritmo de toda la práctica. Esto no es de extrañar ya que el buscar en diversas trayectorias combinado con la capacidad de escapar de óptimos locales de ES forman una dinámica con la cual no sólo no vamos a caer en el primer óptimo local que encontremos, sino que vamos a probar a buscar desde soluciones diversas para poder hacer una exploración aún mayor de todas las soluciones posibles, de la cual obtenemos la mejor solución encontrada.

Al igual que en la práctica anterior, la cuestión más importante es cual de estos es el mejor algoritmo para resolver el problema de MDD. Tras ver y analizar los resultados obtenidos queda claro que el mejor de ellos es ILS-ES, obteniendo el segundo puesto entre todos los algoritmos implementados en la práctica (no consigue mejores resultados que AGE-Uniforme, aunque si tarda mucho menos). También cabe destacar que consigue estos resultados en un tiempo bastante escueto, sobre todo si lo comparamos con la ejecución de cualquiera de los algoritmos genéticos de la práctica anterior, aunque obtiene peores tiempos que el resto de los implementados en esta práctica.

6. Bibliografía

1. Documentación Armadillo. http://arma.sourceforge.net/docs.html
2. Biblioteca de generación de números aleatorios. https://sci2s.ugr.es/node/124