Práctica 2

Problema 1: Técnicas de Búsqueda de Poblaciones para el Problema de la Mínima Dispersión Diferencial (MDD)

UNIVERSIDAD DE GRANADA

Metaheurísticas - Curso 2024/2025
Grupo 1 - Miércoles

David Kessler Martínez
23300373Q
dkesslerm03@correo.ugr.es
e.dkesslerm03@go.ugr.es

Descripción del problema

El problema de la mínima dispersión diferencial (*MDD*) es un problema de optimización combinatoria que se basa seleccionar un subconjunto *M* de m elementos (|*M*| = *m*) de un conjunto inicial *N* de *n* elementos (con *n* > *m*) de forma que se minimice la dispersión entre los elementos escogidos. El objetivo es minimizar:

$$Max_{x_i \in M} \left(\sum_{j \in M} d_{ij} \right) - Min_{x_i \in M} \left(\sum_{j \in M} d_{ij} \right) \text{ con } M \subset N, |M| = m$$

donde:

- *M* es una solución al problema que consiste en un vector que indica la posición en *N* de los *m* elementos seleccionados.
- d_{ij} es la distancia existente entre los elementos *i* y *j*, dada en nuestro caso por una matriz simétrica de tamaño *nxn*.

Este problema tiene aplicaciones prácticas como:

- Diseño de redes: minimizar la diferencia de grados entre nodos puede ayudar a equilibrar la carga y mejorar el rendimiento.
- Redes sociales: al reducir la diferencia de grados entre los nodos, es posible crear una red más equilibrada.
- Gestión de la red eléctrica: optimizar la ubicación y dimensionamiento de los generadores de energía y las líneas de transmisión, minimizando pérdidas de energía.

En nuestro caso, utilizamos 50 casos seleccionados de los conjuntos de instancias disponibles en la librería MDPLIB, todas pertenecientes al grupo GKD, que tienen la estructura;

n m

D

Donde *n* es el número de elementos, *m* es el tamaño de la solución y *D* es la diagonal superior de la matriz de distancias. Cada entrada tiene el formato:

i j d_{ij}

donde *i* y *j* son, respectivamente, la fila y la columna de la matriz *D* y d_{ij} es el valor de la distancia existente entre ellos.

Aplicación de los algoritmos al problema

Hemos implementado siete algoritmos para esta nueva resolución del problema MDD, o al menos, siete variantes de tres algoritmos principales: dos variantes del Algoritmo Genético Generacional, otras dos del Algoritmo Genético Estacionario, y tres variantes del Algoritmo Memético. En la base, todos comparten la misma estructura común heredada de la clase MH, de la que ya se habló en el guión de la práctica anterior, de manera que no se repetirá la explicación. La representación de soluciones y la función objetivo son la misma.

Más concretamente, los algoritmos genéticos heredan de una clase *GeneticAlgorithm* (que a su vez hereda de MH), donde se definen los aspectos comunes a ellos: un constructor, el método de selección mediante torneo, los distintos operadores de cruce, un método de mutación, y un par más de métodos auxiliares. Por su parte, los algoritmos meméticos heredan de la clase *GenerationalGeneticAlgorithm*, ya que realmente se tratan de algoritmos genéticos generacionales a los que se le añade una búsqueda local. Más sobre esto más adelante.

Clase *GeneticAlgorithm*

Vamos a profundizar en la clase *GeneticAlgorithm*. Tiene un atributo privado, un booleano *uniform*, que dicta el operador de cruce a usar en los algoritmos genéticos: si es *true*, se hará un cruce uniforme con reparación, y si es *false*, se hará un cruce de intercambio o posición. El constructor acepta un booleano como parámetro, que inicializa este atributo *uniform*, y tiene el destructor por defecto. También tiene un getter para el atributo *uniform*.

Metiéndonos en materia, el método de selección por torneo acepta como parámetros la población de soluciones y el fitness de cada una de las soluciones de la población, además del número de participaciones del torneo. Devuelve una solución, que es la ganadora del torneo. Esta es la cabecera del método:
tSolution tournament (const vector<tSolution> &poblacion, const vector<tFitness> &fitness, int k)
y hace lo siguiente:

```
n <- tamaño de poblacion
index vector vacío
para todo i desde 0 hasta n-1 hacer:
        index <- i
fin para todo
index <- shuffle(index)
best <- poblacion[index[0]]
best_fitness <- fitness[index[0]]
para todo i desde 1 hasta k, hacer:
        si fitness[index[i]] es menor que best_fitness hacer:
                best <- poblacion[index[i]]
                best_fitness <- fitness[index[i]]
        fin si
fin para todo
devolver best
```

Funciona como se espera, se cogen *k* soluciones de la población y se devuelve la mejor (la que menor fitness tenga). Pasando a los operadores de cruce, el cruce uniforme selecciona los valores que tengan en común ambos padres y obliga que los hijos los tengan. El resto de valores son rellenados aleatoriamente. Este operador necesita reparación porque, como trabajamos con representación binaria, puede ser que al generar los valores aleatorios se tengan más 1s de los que se deberían (ya que necesitamos exactamente *m* 1s). En nuestro caso, simplemente hemos elegido como heurística el barajar las posiciones modificables, y empezar a modificar a partir de ahí. El método acepta como parámetros dos soluciones padres, el tamaño del problema (n) y el tamaño de la solución (m). Devuelve un par <solución, solución>, que es el resultado del cruce de los padres. Esta es la cabecera del método:
pair<tSolution, tSolution> uniformCrossover(const tSolution father1, const tSolution father2, int n, int m)
y hace lo siguiente:

```
// primero pasamos a representación binaria
binary_father1 vector de booleanos de n posiciones inicializado a 0
binary_father2 vector de booleanos de n posiciones inicializado a 0
para cada nodo de father1 hacer:
        binary_father1[nodo] <- true
fin para cada
para cada nodo de father2 hacer:
        binary_father2[nodo] <- true
fin para cada
binary_child1 vector de booleanos de n posiciones inicializado a 0
binary_child2 vector de booleanos de n posiciones inicializado a 0
modifiables vector de enteros vacío

// bucle para crear los hijos
para cada i desde 0 hasta n-1 hacer:
        si binary_father1[i] es igual a binary_father2[i] hacer:
                binary_child1[i] <- binary_father1[i]
                binary_child2[i] <- binary_father2[i]
        en caso contrario hacer:
                binary_child1[i] <- aleatorio(0, 1)
                binary_child2[i] <- aleatorio(0, 1)
                añadir i a modifiables
        fin si
fin para cada

// comprobación para ver si se tiene que hacer reparación
node_count1 <- 0
node_count2 <- 0
para cada i desde 0 hasta n-1 hacer:
        si binary_child1[i] es true hacer node_count1 <- node_count1 + 1
        si binary_child2[i] es true hacer node_count2 <- node_count2 + 1
fin para cada
```

```
// reparacion
modifiables <- shuffle(modifiables)
finished es un booleano

si node_count1 es mayor que m hacer:
        finished <- false
        para cada i desde 0 a modifiables.size - 1 y mientras !finished hacer:
                si binary_child1[modifiables[i]] es true hacer:
                        binary_child1[modifiables[i]] <- false
                        node_count1 <- node_count1 - 1
                        finished <- (node_count1 - m == 0)
                fin si
        fin para cada
en caso contrario, si node_count1 es menor que m hacer:
        finished <- false
        para cada i desde 0 a modifiables.size - 1 y mientras !finished hacer:
                si binary_child1[modifiables[i]] es false hacer:
                        binary_child1[modifiables[i]] <- true
                        node_count1 <- node_count1 + 1
                        finished <- (node_count1 - m == 0)
                fin si
        fin para cada
fin si

// repetimos para el segundo hijo
si node_count2 es mayor que m hacer:
        finished <- false
        para cada i desde 0 a modifiables.size - 1 y mientras !finished hacer:
                si binary_child2[modifiables[i]] es true hacer:
                        binary_child2[modifiables[i]] <- false
                        node_count2 <- node_count2 - 1
                        finished <- (node_count2 - m == 0)
                fin si
        fin para cada
en caso contrario, si node_count2 es menor que m hacer:
        finished <- false
        para cada i desde 0 a modifiables.size - 1 y mientras !finished hacer:
                si binary_child2[modifiables[i]] es false hacer:
                        binary_child2[modifiables[i]] <- true
                        node_count2 <- node_count2 + 1
                        finished <- (node_count2 - m == 0)
                fin si
        fin para cada
fin si

// pasamos a representación entera
child1 <- solucion vacia
child2 <- solucion vacia
```

para cada i desde 0 hasta n-1 hacer:
    si binary_child1[i] es true, entonces añadir i a child1
    si binary_child2[i] es true, entonces añadir i a child2
fin para cada
return child1, child2

El otro operador de cruce es el de intercambio, o posición. Este no necesita reparación, ya que una vez tenemos las posiciones que vienen de los padres, se baraja el resto de bits de cada padre y se distribuyen en cada hijo respectivamente. Esta es la cabecera del método:
pair<tSolution, tSolution> positionCrossover(const tSolution father1, const tSolution father2, int n, int m)
y hace lo siguiente:

// pasamos a representación binaria
binary_father1 vector de booleanos de n posiciones inicializado a 0
binary_father2 vector de booleanos de n posiciones inicializado a 0
para cada nodo de father1 hacer:
    binary_father1[nodo] <- true
fin para cada
para cada nodo de father2 hacer:
    binary_father2[nodo] <- true
fin para cada

binary_child1 vector de booleanos de n posiciones inicializado a false
binary_child2 vector de booleanos de n posiciones inicializado a false
differing_positions vector de enteros vacío
bits1 vector de booleanos vacío
bits2 vector de booleanos vacío

// rellenar las posiciones iguales y tener en cuenta las que cambian
para cada i desde 0 hasta n-1 hacer:
    si binary_father1[i] es igual a binary_father2[i] hacer:
        binary_child1[i] <- binary_father1[i]
        binary_child2[i] <- binary_father2[i]
    en caso contrario hacer:
        añadir i a differing_positions
        añadir binary_father1[i] a bits1
        añadir binary_father2[i] a bits2
    fin si
fin para cada

// se desordenan los valores de cada padre para rellenar los hijos
bits1 <- shuffle(bits1)
bits2 <- shuffle(bits2)
para cada i desde 0 a differing_positions.size-1 hacer:
    pos <- differing_positions[i]
    binary_child1[pos] <- bits1[i]
    binary_child2[pos <- bits2[i]

fin para cada

// pasar a representación entera
child1 <- solucion vacia
child2 <- solucion vacia
para cada i desde 0 hasta n-1 hacer:
    si binary_child1[i] es true, entonces añadir i a child1
    si binary_child2[i] es true, entonces añadir i a child2
fin para cada
return child1, child2

Finalmente, el operador de mutación. Simplemente escoge una solución aleatoria de la población, escoge una posición aleatoria de la misma, y sustituye su valor por un valor que no se encuentre ya en la solución. Su pseudocódigo es el siguiente:

para cada i desde 0 hasta num_mutations-1 hacer:
    pob_size <- tamaño de poblacion
    // elegir una solución aleatoria de la poblacion
    sol_index <- aleatorio(0, pob_size-1)
    &sol <- poblacion[sol_index]

    // elegir nodo a quitar de la solución seleccionada
    sol_size <- tamaño de sol
    pos <- aleatorio(0, sol_size-1)

    // generar un nodo nuevo que no esté ya en la solución
    new_node es un entero
    exists es un booleano
    hacer:
        new_node <- aleatorio(0, n-1)
        exists <- false
        para cada i desde 0 hasta sol_size-1 y mientras !exists hacer:
            si sol[i] es igual a new_node
                exists <- true
            fin si
        fin para cada
    mientras exists sea true

    sol[pos] <- new_node
fin para cada

No se devuelve nada porque se pasa por referencia. Ya lo último son dos métodos auxiliares, que encuentran el mejor y peor índice de una lista de fitness. No se incluyen por su simplicidad y por el límite de páginas (ya me he pasado un poco).

El resto de detalles de implementación de los algoritmos se deja para la siguiente sección. El único cambio que merece la pena mencionar con respecto a la primera práctica es que ahora, la clase *RandomLocalSearch* hereda de *MHTrayectory*, en vez de *MH*, lo que nos

permite hacer la búsqueda local sobre una solución pasada por parámetro, en vez de crear una solución aleatoria. Más sobre esto más adelante.

Estructura algoritmos

Algoritmos Genéticos Generacionales

Los algoritmos genéticos se basan en la mejoría general de una población de soluciones a través de distintas generaciones. Concretamente, los algoritmos genéticos generacionales se caracterizan por cambiar la población completamente entre generaciones, y los algoritmos genéticos estacionarios son aquellos que la van cambiando poco a poco.

El esquema de evolución viene dado por el método *optimize*, como en la práctica anterior. Consiste en las siguientes operaciones, en el siguiente orden: selección, cruce, mutación y reemplazamiento. En el caso del AGG, este método forma parte de la clase *GenerationalGeneticAlgorithm,* que incluye además un método auxiliar *generateChildren* que encapsula la selección, el cruce y la mutación en un solo método. Este es el pseudocódigo del método *optimize*:

```
mdd_problem <- problem // cast necesario solo para obtener el tamaño del problema
n <- tamaño del problema
m <- tamaño de la solución
evals <- 0

//  inicializar poblacion y fitness
poblacion es un vector de soluciones de tamaño 50
para cada i desde 0 a 49 hacer:
        sol es una solucion aleatoria (creada con createSolution)
        añadir sol a poblacion
fin para cada

fitness es un vector de floats de tamaño 50 vacío
best_index <-  -1
best_fitness <-  1000000
para cada i desde 0 hasta poblacion.size hacer:
        fitness[i] <- fitness de la solucion i-ésima de poblacion
        si fitness[i] es menor que best_fitness hacer:
                best_fitness <- fitness[i]
                best_index <- i
        fin si
        evals <- evals + 1
fin para cada

// bucle principal
mientras que evals sea menor que maxevals hacer:
        children <- generateChildren(poblacion, fitness, n, m)
```

```
        // evaluar hijos
        fitness_children es un vector de floats de tamaño 50 vacío
        para cada i desde 0 a 49 hacer:
                fitness_children[i] <- fitness de la solución children[i]
                evals <- evals + 1
        fin para cada

        // elitismo
        worst_index <- findWorst(fitness_children)
        si best_fitness es menor que fitness_children[worst_index] hacer:
                children[worst_index] <- poblacion[best_index]
                fitness_children[worst_index] <- best_fitness
        fin si

        // reemplazo final
        poblacion <- children
        fitness <- fitness_children

        best_index <- findBest(fitness)
        best_fitness <- fitness_children[best_index]
fin mientras

final <- poblacion[best_index]
devolver {final, best_fitness, evals}
```

El método auxiliar que hay que comentar es *generateChildren*. Acepta como parámetros la población, el fitness de la misma, el tamaño del problema y de la solución, y devuelve un vector de soluciones, una nueva población. Este es su pseudocódigo:

```
// selección
fathers es un vector vacío de tamaño 50
para cada i desde 0 hasta 50 hacer:
        fathers[i] <- tournament(poblacion, fitness, 3) // método explicado antes
fin para cada

// cruce
children es un vector vacío de tamaño 50
crossover_num <- 0.7 * 25 // probabilidad de cruce * 25 parejas
para cada i desde 0 hasta 2*crossover_num en intervalos de dos hacer:
        crossed_children es un par de soluciones
        si getUniform() es true hacer:
                crossed_children <- uniformCrossover(fathers[i], fathers[i+1], n, m)
        en caso contrario hacer:
                crossed_children <- positionCrossover(fathers[i], fathers[i+1], n, m)
        fin si
        children[i] <- crossed_children.first
        children[i+1]<- crossed_children.second
fin para cada
```

*para cada i desde 2*crossover_num hasta 50 en intervalos de dos hacer:*
    children[i] <- fathers[i]
fin para cada
// mutación
*num_mutations <- 0.1*50 (probabilidad de cruce * 50 elementos de la poblacion)*
mutatePopulation(children, num_mutations, n)

devolver children

Como habíamos visto, el operador de cruce que se utiliza lo indica el resultado de *getUniform(),* el getter para el atributo *uniform*, inicializado en el constructor. El reemplazamiento con elitismo se produce al final del bucle principal de *optimize*, se ha señalado en el pseudocódigo.

Algoritmos Genéticos Estacionarios

Al igual que en los generacionales, el esquema de evolución viene dado por el método *optimize*, ahora parte de la clase *SteadyStateGeneticAlgorithm*. También usa un método auxiliar *generateChildren* (también de esta clase), que también encapsula la selección, el cruce y la mutación en un solo método.

La diferencia principal entre la implementación de los genéticos y los estacionarios es que en los primeros se trabaja con la población completa de 50 individuos, mientras que en los segundos se trabaja solo con 2 de los 50 individuos. Por supuesto, también hay cambios en el cruce (en el primer caso, se cruzan dos a dos con probabilidad fija, y en el segundo caso, los dos individuos de la población se cruzan siempre) y en la implementación de la mutación. Todas estas diferencias se aprecian mejor observando el pseudocódigo de la función *optimize*:

mdd_problem <- problem // cast necesario solo para obtener el tamaño del problema
n <- tamaño del problema
m <- tamaño de la solución
evals <- 0

// inicializar poblacion y fitness
poblacion es un vector de soluciones de tamaño 50
para cada i desde 0 a 49 hacer:
    sol es una solucion aleatoria (creada con createSolution)
    añadir sol a poblacion
fin para cada

fitness es un vector de floats de tamaño 50 vacío
best_index <- -1
best_fitness <- 1000000
para cada i desde 0 hasta poblacion.size hacer:
    fitness[i] <- fitness de la solucion i-ésima de poblacion

```
        si fitness[i] es menor que best_fitness hacer:
                best_fitness <- fitness[i]
                best_index <- i
        fin si
        evals <- evals + 1
fin para cada

// bucle principal
mientras que evals sea menor que maxevals hacer:
        // generar hijos y evaluarlos
        children <- generateChildren(poblacion, fitness, n, m)
        child1 <- children[0]
        child2 <- children[1]
        fit1 <- fitness(child1)
        fit2 <- fitness(child2)
        evals <- evals + 2

        // comprobar cuál es el mejor
        best_child <- child1
        fit_best_child <- fit1
        si fit2 es menor que fit1 hacer
                best_child <- child2
                fit_best_child <- fit2
        fin si

        // sustituir, si es mejor, el nuevo hijo por el peor individuo de la población anterior
        worst_index <- findWorst(fitness)
        si fit_best_child es menor que fitness[worst_index] hacer:
                poblacion[worst_index] <- best_child
                fitness[worst_index] <- fit_best_child
        fin si

        // actualizar mejores globales
        best_index <- findBest(fitness)
        best_fitness <- fitness[best_index]
fin mientras

final <-poblacion[best_index]
devolver {final, best_fitness, evals}
```

Realmente lo interesante aquí es ver el cambio de trabajar con solo dos individuos frente a la población entera. El cambio principal es en la función *generateChildren:*

```
// torneo para seleccionar los dos padres
father1 <- tournament(poblacion, fitness, 3)
father2 <- tournament(poblacion, fitness, 3)

// cruce obligatorio
```

```
children es un par de soluciones
si getUniform() es true hacer:
        children <- uniformCrossover(father1, father2, n, m)
en caso contrario hacer:
        children <- positionCrossover(father1, father2, n, m)

child1 <- children.first
child2 <- children.second

// individualmente, se decide si mutar cada hijo
si aleatorio(0, 1) es menor que 0.1 hacer:
        children_vector es un vector con child1
        mutatePopulation(children_vector, 1, n)
        child1 <- children_vector[0]
fin si
si aleatorio(0, 1) es menor que 0.1 hacer:
        children_vector es un vector con child2
        mutatePopulation(children_vector, 1, n)
        child2 <- children_vector[0]
fin si

devolver {child1, child2}
```

Igual que el algoritmo genético generacional, el operador de cruce que se utiliza lo indica el resultado de *getUniform(),* el getter para el atributo *uniform*, inicializado en el constructor. Por otra parte, en *optimize* no tenemos un reemplazamiento con elitismo como habíamos visto en los AGG, sino simplemente vemos cuál de los dos hijos que hemos generado es mejor y, si mejora al peor de la generación anterior, se sustituye. Como vimos en clase, otra variante es cambiar los dos hijos, en vez de uno solo, pero en la práctica solo añade complejidad y no mejora los resultados.

Algoritmos Meméticos

Como hemos explicado en la sección anterior, los algoritmos meméticos combinan los algoritmos genéticos generacionales con una búsqueda local. Los genéticos cambian mucho las soluciones, y puede ser que variando un poco las que ya tenemos se obtengan mejores resultados. El principal dilema es el equilibrio entre exploración y explotación, es decir, cada cuánto se aplica una búsqueda local y sobre qué agentes.

Cada 10 generaciones del algoritmo genético generacional le aplicaremos la búsqueda local aleatoria de la primera práctica a tres posibles configuraciones de agentes: todos, el 10% de ellos, o el 10% de los mejores individuos. Hemos decidido hacer que la clase *MemeticAlgorithm* (donde está implementado el método optimize) herede de *GenerationalGeneticAlgorithm* para que pueda usar todos los métodos que hemos explicado en las secciones anteriores.

Como apuntes sobre la estructura de la clase *MemeticAlgorithm*, tiene tres atributos: *strategy*, que es un tipo enumerado de entre *ALL* (búsqueda local a todos los agentes), *SOME_RANDOM* (búsqueda local al 10% de los agentes), *SOME_BEST* (búsqueda local al 10% de los mejores agentes); *period*, un entero que indica cada cuántas generaciones se aplica la búsqueda local (con un valor constante de 10, en nuestro caso) y *pLS*, la proporción de la población sobre la cual se hará la búsqueda local. Tanto *strategy* como *pLS* se inicializan en el constructor que acepta una *strat* de las mencionadas como parámetro, y pLS adopta el valor 0.1 o 1.0 dependiendo del mismo. Este mismo constructor se inicializa con *uniform = true*, es decir, que usamos un algoritmo genético generacional con cruce uniforme, ya que es el cruce que mejores resultados ha obtenido, luego se verá en detalle.

Pasamos a lo importante, la implementación del método *optimize*. Hace lo siguiente:

```
mdd_problem <- problem // cast necesario solo para obtener el tamaño del problema
n <- tamaño del problema
m <- tamaño de la solución
evals <- 0
gen <- 0
bl <- RandomLocalSearch()

//  inicializar poblacion y fitness
poblacion es un vector de soluciones de tamaño 50
para cada i desde 0 a 49 hacer:
        sol es una solucion aleatoria (creada con createSolution)
        añadir sol a poblacion
fin para cada

fitness es un vector de floats de tamaño 50 vacío
best_index <-  -1
best_fitness <-  1000000
para cada i desde 0 hasta poblacion.size hacer:
        fitness[i] <- fitness de la solucion i-ésima de poblacion
        si fitness[i] es menor que best_fitness hacer:
                best_fitness <- fitness[i]
                best_index <- i
        fin si
        evals <- evals + 1
fin para cada

// bucle principal
mientras evals sea menor que maxevals hacer:
        // crear los los hijos y evaluarlos
        gen <- gen + 1
        children <- generateChildren(poblacion, fitness, n, m)
        fitness_children es un vector de floats de tamaño 50 vacío
        para cada i desde 0 a 49 hacer:
                fitness_children[i] <- fitness de la solución children[i]
                evals <- evals + 1
```

```
        fin para cada

        // búsqueda local
        cada 10 generaciones hacer:
                to_improve es un vector de índices vacío
                si strategy es hibridation::ALL hacer:
                        to_improve ahora tiene tamaño 50
                        para cada i desde 0 hasta 49 hacer:
                                to_improve[i] <- i
                en caso contrario hacer:        // SOME_RANDOM y SOME_BEST
                        fitness_index almacena los fitness y su índice
                        para cada i desde 0 a 49 hacer:
                                fitness_index[i] <- {fitness_hijos[i], i}
                        fin para cada
                        si strategy es hibridation::SOME_RANDOM hacer:
                                fitness_index <- shuffle(fitness_index)
                        en caso contrario hacer:
                                fitness_index <- sort(fitness_index)
                        fin si

                        num_to_improve <- pLS * 50
                        para cada i desde 0 a num_to_improve-1 hacer:
                                añadir fitness_index[i].second a to_improve
                        fin para cada
                fin si
                // aplicacion búsqueda local
                para cada i desde 0 a to_improve.size-1 hacer:
                        real_evals <- minimo entre (max_evals - evals, 400)
                        si real_evals > 0 hacer:
                                r <- bl.optimize(problem, children[to_improve[i]],
fitness_hijos[to_improve[i]], real_evals)
                                children[to_improve[i]] <- r.solution
                                fitness_hijos[to_improve[i]] <- r.fitness
                                evals <- evals + r.evaluations
                        fin si
                fin para cada
        fin cada 10 generaciones hacer

        // elitismo
        worst_index <- findWorst(fitness_children)
        si best_fitness es menor que fitness_children[worst_index] hacer:
                children[worst_index] <- poblacion[best_index]
                fitness_children[worst_index] <- best_fitness
        fin si

        // reemplazo final
        poblacion <- children
        fitness <- fitness_children
```

best_index <- findBest(fitness)
best_fitness <- fitness_children[best_index]
fin mientras

final <- poblacion[best_index]
devolver {final, best_fitness, evals}

Como podemos ver, se trata de un AGG con cruce uniforme que simplemente aplica una búsqueda local sobre los hijos generados cada 10 generaciones. Sobre la manera de hacer la búsqueda local, se determinan los agentes sobre la cual hacerse *(to_improve)* en función de strategy, como hemos comentado anteriormente. Finalmente, hay que mencionar que hemos adaptado la clase *RandomLocalSearch* para que herede de *MHTrayectory*. Ahora tenemos las opciones de hacer la búsqueda local de cero (como estaba antes), o a partir de una solución que le pasamos como parámetros (algo que es útil para los meméticos).

En el análisis de resultados vamos a comentar una cosa muy interesante: hemos probado el algoritmo memético con AGG cruce de uniforme y AGG cruce de posición y han salido resultados distintos a los esperados!

Estructura del código

El proyecto sigue la plantilla de PRADO. Se va a explicar directorio a directorio, indicando los cambios que hemos realizado. **En negrita están puestos los cambios de esta sección respecto a la práctica anterior.**

El directorio *common* incluye, como su nombre indica, archivos comunes a los dos problemas. De entre ellos, hemos modificado únicamente *solution.h* (para modificar los tipos de datos acorde a nuestro problema) y *problem.h* (para poner público el destructor de la clase *SolutionFactoringInfo*, aunque al final no hemos usado esta clase para nuestra resolución). **También nuevo para esta práctica 2, hemos añadido *#pragma once* en *mhtrayectory.h*.**

El directorio *data* lo hemos creado nosotros para incluir en el proyecto los 50 archivos de datos sobre los que se trabaja en el main.

El directorio *inc*, incluido en la plantilla, recoge los archivos .h de la implementación. Esto es:

- ***GA.h***
- ***GGA.h***
- *heuristicLS.h*
- ***MA.h***
- *mddp.h*
- *randomLS.h*
- ***SGA.h***

En cada archivo está la declaración de las clases y métodos usados para resolver el problema. *En mddp.h* está definido todo lo relacionado al problema MDD, **y el resto de archivos definen lo necesario para cada algoritmo, según su nombre.**

El directorio *src*, incluido en la plantilla, recoge los archivos .cpp de la implementación. Esto es:

- ***GA.cpp***
- ***GGA.cpp***
- *heuristicLS.cpp*
- ***MA.cpp***
- *mddp.cpp*
- *randomLS.cpp*
- ***SGA.cpp***

En cada archivo está la implementación de las clases y métodos usados para resolver el problema. En *mddp.cpp* está implementado todo lo relacionado al problema MDD, **en *GA.cpp* está implementado todo lo común a AGGs y AGEs, como hemos explicado en la sección correspondiente. Finalmente, *GGA.cpp, SGA.cpp, heuristicLS.cpp* y *randomLS.cpp* contienen el *optimize* respectivo para cada algoritmo.**

Aparte del *main*, hemos creado otro *cpp* que no se agrupa en ninguna carpeta. Este es *savetofile.cpp*, que como su nombre indica, guarda en un archivo de texto los resultados de la ejecución de las **7 variaciones de los 3 algoritmos** con 5 semillas distintas, para todos los archivos de datos del directorio *data*, junto con el tiempo que tarda cada uno. Esto se ha utilizado para el paso a tablas, a la hora de comparar y analizar los resultados para cada algoritmo.

Finalmente tenemos *main.cpp*, que muestra por pantalla o bien el resultado de las ejecuciones, con el archivo de datos pasado por terminal, con las 5 semillas predefinidas, o bien el resultado de las ejecuciones con la semilla indicada por terminal. Se usa una función auxiliar, *print_result*, que imprime los resultados de una ejecución.

Manual de ejecución

Para compilar el proyecto, se ha utilizado el Makefile incluido en la plantilla. Lo único que se ha cambiado es el *CMakeLists.txt*, añadiendo una línea para poder ejecutar *savetofile.cpp* cuando nos interese independientemente del main. Para generar los ejecutables, simplemente hay que hacer *cmake .* en la carpeta del proyecto, y a continuación, *make*. Esto genera *main* y *savetofile*, dos archivos que se ejecutan:

- *./main data/<archivo> <seed>*
- *./savetofile*

Los dos parámetros que acepta el *main* son el archivo de datos sobre el que probar los algoritmos (se encuentra en la carpeta *data/*) y, opcionalmente, la semilla a utilizar para la generación de números aleatorios. Digo opcionalmente porque en caso de no incluirla (y simplemente ejecutar como ./main data/<archivo>) se utilizan las 5 semillas definidas por defecto: {42, 0, 31415, 123, 2025}. **Esta es la ejecución que se recomienda para la prueba del programa.**

Un ejemplo de ejecución sería simplemente *./main data/GKD-b_7_n25_m7.txt* , que utilizaría las semillas por defecto, o *./main data/GKD_b_7_n25_m7.txt 42*, para ejecutar únicamente la semilla 42. Para el otro ejecutable, simplemente *./savetofile* genera el txt con los resultados.

Experimentos y análisis de resultados

Descripción

El problema se ha evaluado sobre los 50 archivos de datos, cuyo formato explicamos en el primer apartado. Cada versión del algoritmo implementado (*AGG-uniforme, AGG-posición, AGE-uniforme, AGE-posición, AM-(10, 1.0), AM-(10, 0.1random) y AM-(10, 0.1best)*, y **EXTRA: los algoritmos meméticos con cruce de posición**) se ejecuta 5 veces sobre cada archivo de datos, una con cada semilla de {42, 0, 31415, 123, 2025}. Los valores de fitness, y tiempos de ejecución, son el resultado de ejecutar *./savetofile*, que los almacena en *results.txt.* Luego los he pasado al archivo excel de PRADO, y son los resultados que vamos a analizar.

Debemos justificar por qué hemos incluido también los algoritmos meméticos con el cruce de posición. Según el guión de la práctica, debíamos solo hacer los algoritmos meméticos con el cruce que mejores resultados obtenga en AGG, y en nuestro caso ese es el cruce uniforme. Esto es lo que hice al principio, pero cuando estaba viendo los resultados obtenidos con ambos cruces en los AGGs vi que no había casi diferencia (más adelante lo analizamos todo, por supuesto), y sabía que el cruce de posición tardaba menos tiempo que el uniforme, de manera que me pareció interesante probar cómo funcionaban los meméticos con el cruce de posición, solo por curiosidad y sin intención de incluirlo en la memoria, ni en la práctica en general.
Pensaba que como los AGGs eran casi iguales con los dos operadores de cruce, también lo serían los AMs asociados, pero al ver que uno era considerablemente más preciso y rápido que el otro, decidí incluirlo como apartado extra en esta memoria. Ahora sí, pasamos a mostrar los resultados y a su análisis.

Resultados

Algoritmo Genético Generacional (Cruce uniforme)

Algoritmo Genético Generacional (Cruce uniforme)			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
GKD-b_1_n25_m2	0,0000	0,00	7,00E-01
GKD-b_2_n25_m2	0,0000	0,00	6,98E-01
GKD-b_3_n25_m2	0,0000	0,00	7,59E-01
GKD-b_4_n25_m2	0,0000	0,00	7,03E-01
GKD-b_5_n25_m2	0,0000	0,00	7,13E-01
GKD-b_6_n25_m7	24	46,97	8,41E-01
GKD-b_7_n25_m7	28,1489	49,91	7,84E-01
GKD-b_8_n25_m7	25,4796	34,22	8,75E-01
GKD-b_9_n25_m7	34,4656	50,47	7,78E-01
GKD-b_10_n25_m7	31,1221	25,25	8,00E-01
GKD-b_11_n50_m5	13,8818	86,12	1,10E+00
GKD-b_12_n50_m5	14,3203	85,19	1,06E+00
GKD-b_13_n50_m5	11,4137	79,30	1,04E+00
GKD-b_14_n50_m5	13,3536	87,54	1,03E+00
GKD-b_15_n50_m5	10,9285	73,89	1,04E+00
GKD-b_16_n50_m15	94,4565	54,75	1,36E+00
GKD-b_17_n50_m15	81,6161	41,06	1,36E+00
GKD-b_18_n50_m15	95,3722	54,71	1,25E+00
GKD-b_19_n50_m15	85,4967	45,71	1,20E+00
GKD-b_20_n50_m15	113,5890	57,99	1,21E+00
GKD-b_21_n100_m10	31,4542	56,02	1,58E+00
GKD-b_22_n100_m10	34,5770	60,48	1,57E+00
GKD-b_23_n100_m10	40,1668	61,80	1,62E+00
GKD-b_24_n100_m10	33,1477	73,93	1,61E+00
GKD-b_25_n100_m10	35,6832	51,80	1,62E+00
GKD-b_26_n100_m30	354,4000	52,39	2,46E+00
GKD-b_27_n100_m30	300,5730	57,72	2,46E+00
GKD-b_28_n100_m30	262,9790	59,55	2,70E+00
GKD-b_29_n100_m30	263,3100	47,80	2,62E+00
GKD-b_30_n100_m30	344,0330	62,95	2,24E+00
GKD-b_31_n125_m12	41,8432	71,93	2,18E+00
GKD-b_32_n125_m12	45,2739	58,50	2,08E+00
GKD-b_33_n125_m12	51,1408	63,76	1,86E+00
GKD-b_34_n125_m12	47,1817	58,70	1,95E+00
GKD-b_35_n125_m12	39,6919	54,37	2,01E+00
GKD-b_36_n125_m37	319,9070	51,41	2,78E+00
GKD-b_37_n125_m37	454,0900	56,20	2,78E+00
GKD-b_38_n125_m37	483,4580	61,12	2,81E+00
GKD-b_39_n125_m37	420,9940	59,95	2,77E+00
GKD-b_40_n125_m37	459,2740	61,20	2,73E+00
GKD-b_41_n150_m15	61,4949	62,04	2,13E+00
GKD-b_42_n150_m15	63,3770	57,73	2,14E+00
GKD-b_43_n150_m15	75,0448	64,35	2,10E+00
GKD-b_44_n150_m15	54,6269	52,52	2,14E+00
GKD-b_45_n150_m15	62,8776	55,83	2,15E+00
GKD-b_46_n150_m45	587,4410	61,23	3,45E+00
GKD-b_47_n150_m45	522,7300	56,27	3,45E+00
GKD-b_48_n150_m45	492,7970	53,99	3,43E+00
GKD-b_49_n150_m45	493,0930	54,08	3,46E+00
GKD-b_50_n150_m45	549,8520	54,74	3,43E+00

Media Desv:	**52,55**
Media Tiempo:	**1,83E+00**

Algoritmo Genético Generacional (Cruce de posición)

Algoritmo Genético Generacional (Cruce de posición)			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
GKD-b_1_n25_m2	0,0000	0,00	6,49E-01
GKD-b_2_n25_m2	0,0000	0,00	6,73E-01
GKD-b_3_n25_m2	0,0000	0,00	6,58E-01
GKD-b_4_n25_m2	0,0000	0,00	6,51E-01
GKD-b_5_n25_m2	0,0000	0,00	6,62E-01
GKD-b_6_n25_m7	28	54,02	7,17E-01
GKD-b_7_n25_m7	27,8581	49,39	7,00E-01
GKD-b_8_n25_m7	34,9109	51,99	6,93E-01
GKD-b_9_n25_m7	30,8153	44,61	6,87E-01
GKD-b_10_n25_m7	35,8736	35,15	6,95E-01
GKD-b_11_n50_m5	10,3051	81,31	8,68E-01
GKD-b_12_n50_m5	8,8495	76,03	8,68E-01
GKD-b_13_n50_m5	11,2554	79,01	8,66E-01
GKD-b_14_n50_m5	11,3862	85,39	8,79E-01
GKD-b_15_n50_m5	11,9554	76,14	8,66E-01
GKD-b_16_n50_m15	96,1437	55,54	1,07E+00
GKD-b_17_n50_m15	91,9444	47,68	1,07E+00
GKD-b_18_n50_m15	97,5874	55,74	1,07E+00
GKD-b_19_n50_m15	105,5000	56,01	1,06E+00
GKD-b_20_n50_m15	113,1170	57,82	1,09E+00
GKD-b_21_n100_m10	32,8054	57,84	1,36E+00
GKD-b_22_n100_m10	36,5814	62,65	1,36E+00
GKD-b_23_n100_m10	35,5102	56,79	1,37E+00
GKD-b_24_n100_m10	35,6575	75,77	1,38E+00
GKD-b_25_n100_m10	32,4914	47,06	1,36E+00
GKD-b_26_n100_m30	343,6300	50,90	1,99E+00
GKD-b_27_n100_m30	302,6370	58,00	1,95E+00
GKD-b_28_n100_m30	295,3950	63,99	1,97E+00
GKD-b_29_n100_m30	293,8500	53,22	1,97E+00
GKD-b_30_n100_m30	327,9040	61,12	1,95E+00
GKD-b_31_n125_m12	41,3371	71,59	1,61E+00
GKD-b_32_n125_m12	41,3173	54,53	1,57E+00
GKD-b_33_n125_m12	58,8614	68,52	1,61E+00
GKD-b_34_n125_m12	56,4781	65,49	1,57E+00
GKD-b_35_n125_m12	43,2080	58,08	1,58E+00
GKD-b_36_n125_m37	358,2590	56,61	2,47E+00
GKD-b_37_n125_m37	416,6500	52,26	2,50E+00
GKD-b_38_n125_m37	491,5430	61,76	2,49E+00
GKD-b_39_n125_m37	398,1660	57,66	2,46E+00
GKD-b_40_n125_m37	415,1560	57,08	2,49E+00
GKD-b_41_n150_m15	51,4198	54,60	1,87E+00
GKD-b_42_n150_m15	73,4639	63,53	1,85E+00
GKD-b_43_n150_m15	72,8902	63,29	1,89E+00
GKD-b_44_n150_m15	60,1649	56,89	1,90E+00
GKD-b_45_n150_m15	66,9063	58,49	1,86E+00
GKD-b_46_n150_m45	569,3210	60,00	3,12E+00
GKD-b_47_n150_m45	482,6610	52,64	3,14E+00
GKD-b_48_n150_m45	488,3040	53,56	3,13E+00
GKD-b_49_n150_m45	501,3520	54,84	3,14E+00
GKD-b_50_n150_m45	576,3260	56,82	3,13E+00

Media Desv:	**53,43**
Media Tiempo:	**1,57E+00**

Algoritmo Genético Estacionario (Cruce uniforme)

Algoritmo Genético Estacionario (Cruce uniforme)			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
GKD-b_1_n25_m2	0,0000	0,00	8,72E-01
GKD-b_2_n25_m2	0,0000	0,00	8,72E-01
GKD-b_3_n25_m2	0,0000	0,00	8,69E-01
GKD-b_4_n25_m2	0,0000	0,00	8,74E-01
GKD-b_5_n25_m2	0,0000	0,00	8,69E-01
GKD-b_6_n25_m7	32	60,09	9,57E-01
GKD-b_7_n25_m7	36,0716	60,91	9,52E-01
GKD-b_8_n25_m7	27,3517	38,72	9,58E-01
GKD-b_9_n25_m7	37,3174	54,26	9,59E-01
GKD-b_10_n25_m7	29,7516	21,80	9,55E-01
GKD-b_11_n50_m5	11,5512	83,33	1,25E+00
GKD-b_12_n50_m5	12,1893	82,60	1,25E+00
GKD-b_13_n50_m5	17,3034	86,35	1,25E+00
GKD-b_14_n50_m5	9,9516	83,29	1,25E+00
GKD-b_15_n50_m5	11,9825	76,19	1,26E+00
GKD-b_16_n50_m15	84,1563	49,21	1,49E+00
GKD-b_17_n50_m15	79,8392	39,74	1,49E+00
GKD-b_18_n50_m15	101,5370	57,46	1,48E+00
GKD-b_19_n50_m15	93,7831	50,51	1,48E+00
GKD-b_20_n50_m15	106,0920	55,02	1,49E+00
GKD-b_21_n100_m10	40,7876	66,09	2,03E+00
GKD-b_22_n100_m10	41,0624	66,72	2,04E+00
GKD-b_23_n100_m10	33,0500	53,57	2,02E+00
GKD-b_24_n100_m10	37,6072	77,02	2,22E+00
GKD-b_25_n100_m10	44,5581	61,40	2,03E+00
GKD-b_26_n100_m30	293,9300	42,60	2,69E+00
GKD-b_27_n100_m30	356,4230	64,34	2,69E+00
GKD-b_28_n100_m30	310,7640	65,77	2,68E+00
GKD-b_29_n100_m30	351,4420	60,89	2,68E+00
GKD-b_30_n100_m30	315,0480	59,54	2,69E+00
GKD-b_31_n125_m12	43,7204	73,14	2,40E+00
GKD-b_32_n125_m12	43,0115	56,32	2,41E+00
GKD-b_33_n125_m12	53,5204	65,37	2,39E+00
GKD-b_34_n125_m12	47,7372	59,18	2,39E+00
GKD-b_35_n125_m12	46,9652	61,43	2,41E+00
GKD-b_36_n125_m37	309,5390	49,79	3,38E+00
GKD-b_37_n125_m37	409,5300	51,43	3,37E+00
GKD-b_38_n125_m37	387,3780	51,48	3,37E+00
GKD-b_39_n125_m37	377,6680	55,36	3,38E+00
GKD-b_40_n125_m37	472,7640	62,31	3,38E+00
GKD-b_41_n150_m15	57,2323	59,21	2,80E+00
GKD-b_42_n150_m15	76,6592	65,05	2,79E+00
GKD-b_43_n150_m15	71,8929	62,79	2,78E+00
GKD-b_44_n150_m15	55,0137	52,86	2,80E+00
GKD-b_45_n150_m15	62,7751	55,76	2,80E+00
GKD-b_46_n150_m45	601,3040	62,12	4,16E+00
GKD-b_47_n150_m45	517,0820	55,79	4,18E+00
GKD-b_48_n150_m45	506,8120	55,26	4,15E+00
GKD-b_49_n150_m45	544,2280	58,40	4,19E+00
GKD-b_50_n150_m45	533,3420	53,34	4,14E+00

Media Desv:	**53,68**
Media Tiempo:	**2,21E+00**

Algoritmo Genético Estacionario (Cruce de posición)

Algoritmo Genético Estacionario (Cruce de posición)			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
GKD-b_1_n25_m2	0,0000	0,00	8,70E-01
GKD-b_2_n25_m2	0,0000	0,00	8,66E-01
GKD-b_3_n25_m2	0,0000	0,00	8,69E-01
GKD-b_4_n25_m2	0,0000	0,00	8,63E-01
GKD-b_5_n25_m2	0,0000	0,00	8,63E-01
GKD-b_6_n25_m7	25	49,67	8,87E-01
GKD-b_7_n25_m7	30,8945	54,36	8,89E-01
GKD-b_8_n25_m7	32,8249	48,94	8,94E-01
GKD-b_9_n25_m7	34,5863	50,65	9,03E-01
GKD-b_10_n25_m7	35,1721	33,85	8,93E-01
GKD-b_11_n50_m5	14,9328	87,10	1,13E+00
GKD-b_12_n50_m5	14,1356	85,00	1,12E+00
GKD-b_13_n50_m5	14,5355	83,75	1,12E+00
GKD-b_14_n50_m5	13,4901	87,67	1,12E+00
GKD-b_15_n50_m5	11,7569	75,73	1,12E+00
GKD-b_16_n50_m15	79,1824	46,02	1,34E+00
GKD-b_17_n50_m15	107,5400	55,27	1,35E+00
GKD-b_18_n50_m15	93,9143	54,00	1,35E+00
GKD-b_19_n50_m15	87,5556	46,99	1,35E+00
GKD-b_20_n50_m15	91,7597	48,00	1,35E+00
GKD-b_21_n100_m10	37,6032	63,22	1,75E+00
GKD-b_22_n100_m10	36,1987	62,25	1,75E+00
GKD-b_23_n100_m10	36,2620	57,68	1,75E+00
GKD-b_24_n100_m10	33,0090	73,82	1,75E+00
GKD-b_25_n100_m10	33,9532	49,34	1,76E+00
GKD-b_26_n100_m30	352,6870	52,16	2,41E+00
GKD-b_27_n100_m30	323,7740	60,75	2,40E+00
GKD-b_28_n100_m30	302,9190	64,88	2,40E+00
GKD-b_29_n100_m30	286,5600	52,03	2,41E+00
GKD-b_30_n100_m30	358,1440	64,41	2,40E+00
GKD-b_31_n125_m12	39,8895	70,56	2,05E+00
GKD-b_32_n125_m12	44,6389	57,91	2,06E+00
GKD-b_33_n125_m12	39,3190	52,87	2,07E+00
GKD-b_34_n125_m12	42,2450	53,87	2,06E+00
GKD-b_35_n125_m12	61,5803	70,59	2,06E+00
GKD-b_36_n125_m37	367,0640	57,65	3,01E+00
GKD-b_37_n125_m37	408,7110	51,34	3,02E+00
GKD-b_38_n125_m37	407,3650	53,86	3,03E+00
GKD-b_39_n125_m37	401,8610	58,05	3,02E+00
GKD-b_40_n125_m37	374,6490	52,44	3,01E+00
GKD-b_41_n150_m15	58,8353	60,32	2,39E+00
GKD-b_42_n150_m15	71,8165	62,70	2,38E+00
GKD-b_43_n150_m15	75,4119	64,52	2,39E+00
GKD-b_44_n150_m15	51,7806	49,91	2,38E+00
GKD-b_45_n150_m15	64,5724	56,99	2,39E+00
GKD-b_46_n150_m45	567,2830	59,85	3,73E+00
GKD-b_47_n150_m45	487,0660	53,07	3,74E+00
GKD-b_48_n150_m45	472,1730	51,98	3,71E+00
GKD-b_49_n150_m45	558,2380	59,44	3,70E+00
GKD-b_50_n150_m45	633,3540	60,71	3,72E+00

Media Desv:	**53,32**
Media Tiempo:	**1,96E+00**

Algoritmo Memético (AM-(10, 1.0))

Algoritmo Memético (ALL)			
Caso	Coste medio obtenido	Desv	Tiempo (s)
GKD-b_1_n25_m2	0,0000	0,00	1,67E-01
GKD-b_2_n25_m2	0,0000	0,00	1,65E-01
GKD-b_3_n25_m2	0,0000	0,00	1,65E-01
GKD-b_4_n25_m2	0,0000	0,00	1,68E-01
GKD-b_5_n25_m2	0,0000	0,00	1,66E-01
GKD-b_6_n25_m7	15	12,67	1,11E-01
GKD-b_7_n25_m7	15,7306	10,37	1,18E-01
GKD-b_8_n25_m7	16,7610	0,00	1,11E-01
GKD-b_9_n25_m7	18,6582	8,52	1,10E-01
GKD-b_10_n25_m7	25,0488	7,12	1,10E-01
GKD-b_11_n50_m5	5,3450	63,96	9,43E-02
GKD-b_12_n50_m5	5,8392	63,68	9,42E-02
GKD-b_13_n50_m5	5,6541	58,22	9,32E-02
GKD-b_14_n50_m5	4,5422	63,38	9,39E-02
GKD-b_15_n50_m5	5,2126	45,27	9,38E-02
GKD-b_16_n50_m15	57,6693	25,88	1,47E-01
GKD-b_17_n50_m15	63,9445	24,77	1,52E-01
GKD-b_18_n50_m15	61,1873	29,40	1,44E-01
GKD-b_19_n50_m15	68,0882	31,83	1,49E-01
GKD-b_20_n50_m15	65,3702	27,01	1,49E-01
GKD-b_21_n100_m10	27,1334	49,02	1,66E-01
GKD-b_22_n100_m10	26,7463	48,91	1,64E-01
GKD-b_23_n100_m10	26,8200	42,78	1,68E-01
GKD-b_24_n100_m10	27,2073	68,24	1,53E-01
GKD-b_25_n100_m10	26,1715	34,28	1,60E-01
GKD-b_26_n100_m30	235,4640	28,34	4,40E-01
GKD-b_27_n100_m30	234,7030	45,85	4,65E-01
GKD-b_28_n100_m30	214,2740	50,35	4,75E-01
GKD-b_29_n100_m30	200,6240	31,49	4,53E-01
GKD-b_30_n100_m30	197,9820	35,61	4,62E-01
GKD-b_31_n125_m12	28,4877	58,77	2,46E-01
GKD-b_32_n125_m12	33,9679	44,69	2,00E-01
GKD-b_33_n125_m12	36,0611	48,61	2,17E-01
GKD-b_34_n125_m12	36,1081	46,03	2,23E-01
GKD-b_35_n125_m12	32,5842	44,41	2,18E-01
GKD-b_36_n125_m37	248,0130	37,33	6,50E-01
GKD-b_37_n125_m37	343,1960	42,05	7,74E-01
GKD-b_38_n125_m37	327,2840	42,57	7,61E-01
GKD-b_39_n125_m37	266,0650	36,64	7,71E-01
GKD-b_40_n125_m37	325,3450	45,23	6,79E-01
GKD-b_41_n150_m15	47,7555	51,11	3,13E-01
GKD-b_42_n150_m15	51,3230	47,80	3,09E-01
GKD-b_43_n150_m15	53,4440	49,94	3,17E-01
GKD-b_44_n150_m15	44,0358	41,10	2,86E-01
GKD-b_45_n150_m15	44,5388	37,64	3,18E-01
GKD-b_46_n150_m45	435,1880	47,67	1,07E+00
GKD-b_47_n150_m45	403,0660	43,28	1,11E+00
GKD-b_48_n150_m45	362,1240	37,38	1,07E+00
GKD-b_49_n150_m45	445,4970	49,18	1,06E+00
GKD-b_50_n150_m45	433,6440	42,61	1,05E+00

Media Desv:	**36,02**
Media Tiempo:	**3,47E-01**

Algoritmo Memético (AM-(10, 0.1random))

Algoritmo Memético (SOME_RANDOM)			
Caso	Coste medio obtenido	Desv	Tiempo (s)
GKD-b_1_n25_m2	0,0000	0,00	4,72E-01
GKD-b_2_n25_m2	0,0000	0,00	4,66E-01
GKD-b_3_n25_m2	0,0000	0,00	4,67E-01
GKD-b_4_n25_m2	0,0000	0,00	4,68E-01
GKD-b_5_n25_m2	0,0000	0,00	4,72E-01
GKD-b_6_n25_m7	15	15,22	3,51E-01
GKD-b_7_n25_m7	16,3979	14,02	3,53E-01
GKD-b_8_n25_m7	18,7678	10,69	3,55E-01
GKD-b_9_n25_m7	21,7656	21,58	3,52E-01
GKD-b_10_n25_m7	25,0488	7,12	3,51E-01
GKD-b_11_n50_m5	4,9470	61,07	3,22E-01
GKD-b_12_n50_m5	6,4776	67,26	3,22E-01
GKD-b_13_n50_m5	6,4780	63,53	3,25E-01
GKD-b_14_n50_m5	6,6460	74,97	3,25E-01
GKD-b_15_n50_m5	7,3346	61,10	3,21E-01
GKD-b_16_n50_m15	54,9933	22,27	3,10E-01
GKD-b_17_n50_m15	74,3805	35,32	3,13E-01
GKD-b_18_n50_m15	71,0242	39,18	3,08E-01
GKD-b_19_n50_m15	66,4163	30,12	3,10E-01
GKD-b_20_n50_m15	71,9817	33,71	3,09E-01
GKD-b_21_n100_m10	31,6963	56,36	3,84E-01
GKD-b_22_n100_m10	27,6024	50,50	3,91E-01
GKD-b_23_n100_m10	28,1004	45,39	3,85E-01
GKD-b_24_n100_m10	28,1770	69,33	3,84E-01
GKD-b_25_n100_m10	31,3288	45,10	3,86E-01
GKD-b_26_n100_m30	256,2890	34,16	6,24E-01
GKD-b_27_n100_m30	206,1510	38,35	6,29E-01
GKD-b_28_n100_m30	258,5940	58,86	6,18E-01
GKD-b_29_n100_m30	202,3400	32,07	6,24E-01
GKD-b_30_n100_m30	227,5380	43,97	6,21E-01
GKD-b_31_n125_m12	23,6758	50,39	4,66E-01
GKD-b_32_n125_m12	36,2178	48,12	4,66E-01
GKD-b_33_n125_m12	37,5132	50,60	4,65E-01
GKD-b_34_n125_m12	39,1013	50,16	4,67E-01
GKD-b_35_n125_m12	34,8661	48,05	4,64E-01
GKD-b_36_n125_m37	273,8480	43,24	8,10E-01
GKD-b_37_n125_m37	345,4320	42,42	8,31E-01
GKD-b_38_n125_m37	344,3260	45,41	8,07E-01
GKD-b_39_n125_m37	289,6100	41,79	8,10E-01
GKD-b_40_n125_m37	323,0610	44,84	7,86E-01
GKD-b_41_n150_m15	50,0118	53,32	5,46E-01
GKD-b_42_n150_m15	50,8889	47,36	5,50E-01
GKD-b_43_n150_m15	52,8127	49,34	5,56E-01
GKD-b_44_n150_m15	48,2788	46,28	5,53E-01
GKD-b_45_n150_m15	52,4011	47,00	5,61E-01
GKD-b_46_n150_m45	459,2670	50,41	1,06E+00
GKD-b_47_n150_m45	396,3520	42,32	1,06E+00
GKD-b_48_n150_m45	396,1230	42,76	1,04E+00
GKD-b_49_n150_m45	460,5390	50,84	1,04E+00
GKD-b_50_n150_m45	407,5090	38,93	1,04E+00

Media Desv:	**39,30**
Media Tiempo:	**5,34E-01**

Algoritmo Memético (AM-(10, 0.1best))

Algoritmo Memético (SOME_BEST)			
Caso	Coste medio obtenido	Desv	Tiempo (s)
GKD-b_1_n25_m2	0,0000	0,00	1,61E-01
GKD-b_2_n25_m2	0,0000	0,00	1,60E-01
GKD-b_3_n25_m2	0,0000	0,00	1,59E-01
GKD-b_4_n25_m2	0,0000	0,00	1,59E-01
GKD-b_5_n25_m2	0,0000	0,00	1,59E-01
GKD-b_6_n25_m7	13	2,34	1,06E-01
GKD-b_7_n25_m7	15,7869	10,69	1,08E-01
GKD-b_8_n25_m7	17,6006	4,77	1,09E-01
GKD-b_9_n25_m7	17,6536	3,31	1,07E-01
GKD-b_10_n25_m7	25,2613	7,90	1,09E-01
GKD-b_11_n50_m5	4,9024	60,71	9,38E-02
GKD-b_12_n50_m5	4,1818	49,28	9,48E-02
GKD-b_13_n50_m5	5,0815	53,51	9,05E-02
GKD-b_14_n50_m5	6,0200	72,37	9,24E-02
GKD-b_15_n50_m5	5,1097	44,16	8,98E-02
GKD-b_16_n50_m15	60,7820	29,67	1,39E-01
GKD-b_17_n50_m15	73,4011	34,46	1,48E-01
GKD-b_18_n50_m15	65,4420	33,99	1,37E-01
GKD-b_19_n50_m15	63,6268	27,06	1,37E-01
GKD-b_20_n50_m15	69,5645	31,41	1,39E-01
GKD-b_21_n100_m10	25,7933	46,37	1,53E-01
GKD-b_22_n100_m10	24,3622	43,91	1,48E-01
GKD-b_23_n100_m10	24,1405	36,43	1,57E-01
GKD-b_24_n100_m10	24,4893	64,72	1,44E-01
GKD-b_25_n100_m10	26,7979	35,81	1,52E-01
GKD-b_26_n100_m30	240,1760	29,75	4,08E-01
GKD-b_27_n100_m30	232,9950	45,45	4,31E-01
GKD-b_28_n100_m30	191,5090	44,45	4,51E-01
GKD-b_29_n100_m30	193,8300	29,09	4,23E-01
GKD-b_30_n100_m30	203,2100	37,27	4,14E-01
GKD-b_31_n125_m12	25,1831	53,36	2,22E-01
GKD-b_32_n125_m12	33,6197	44,11	1,95E-01
GKD-b_33_n125_m12	28,9562	36,00	2,02E-01
GKD-b_34_n125_m12	39,5452	50,72	2,06E-01
GKD-b_35_n125_m12	31,9777	43,36	1,94E-01
GKD-b_36_n125_m37	261,1510	40,48	5,85E-01
GKD-b_37_n125_m37	321,0170	38,04	7,24E-01
GKD-b_38_n125_m37	351,2810	46,49	6,52E-01
GKD-b_39_n125_m37	265,8940	36,59	7,03E-01
GKD-b_40_n125_m37	311,7540	42,84	6,14E-01
GKD-b_41_n150_m15	43,0689	45,79	2,83E-01
GKD-b_42_n150_m15	51,1598	47,64	2,89E-01
GKD-b_43_n150_m15	43,6805	38,75	2,98E-01
GKD-b_44_n150_m15	46,3778	44,08	2,82E-01
GKD-b_45_n150_m15	49,0789	43,41	2,67E-01
GKD-b_46_n150_m45	429,8830	47,02	9,92E-01
GKD-b_47_n150_m45	395,1350	42,15	1,03E+00
GKD-b_48_n150_m45	342,3940	33,78	9,50E-01
GKD-b_49_n150_m45	442,6540	48,85	9,48E-01
GKD-b_50_n150_m45	406,9090	38,84	9,80E-01

Media Desv:	**34,82**
Media Tiempo:	**3,20E-01**

EXTRA: Algoritmo Memético (AM-(10, 1.0)) (Cruce de posición)

Algoritmo Memético (ALL) con cruce de posición			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
GKD-b_1_n25_m2	0,0000	0,00	1,58E-01
GKD-b_2_n25_m2	0,0000	0,00	1,57E-01
GKD-b_3_n25_m2	0,0000	0,00	1,59E-01
GKD-b_4_n25_m2	0,0000	0,00	1,57E-01
GKD-b_5_n25_m2	0,0000	0,00	1,59E-01
GKD-b_6_n25_m7	14	9,48	1,04E-01
GKD-b_7_n25_m7	15,4205	8,57	1,06E-01
GKD-b_8_n25_m7	18,6137	9,95	1,05E-01
GKD-b_9_n25_m7	21,2863	19,81	1,03E-01
GKD-b_10_n25_m7	24,4542	4,86	1,05E-01
GKD-b_11_n50_m5	5,6006	65,61	8,93E-02
GKD-b_12_n50_m5	4,7000	54,87	8,97E-02
GKD-b_13_n50_m5	6,6130	64,28	8,81E-02
GKD-b_14_n50_m5	3,7450	55,59	8,90E-02
GKD-b_15_n50_m5	5,5773	48,84	8,82E-02
GKD-b_16_n50_m15	56,8107	24,76	1,47E-01
GKD-b_17_n50_m15	63,7856	24,58	1,44E-01
GKD-b_18_n50_m15	57,3125	24,63	1,40E-01
GKD-b_19_n50_m15	61,1943	24,16	1,40E-01
GKD-b_20_n50_m15	69,2948	31,14	1,36E-01
GKD-b_21_n100_m10	26,0127	46,83	1,47E-01
GKD-b_22_n100_m10	23,4742	41,79	1,52E-01
GKD-b_23_n100_m10	26,7254	42,58	1,47E-01
GKD-b_24_n100_m10	26,7751	67,73	1,42E-01
GKD-b_25_n100_m10	27,3665	37,15	1,48E-01
GKD-b_26_n100_m30	232,5270	27,44	4,14E-01
GKD-b_27_n100_m30	213,0260	40,34	4,39E-01
GKD-b_28_n100_m30	203,2170	47,65	4,52E-01
GKD-b_29_n100_m30	206,7040	33,50	4,24E-01
GKD-b_30_n100_m30	223,8320	43,05	4,33E-01
GKD-b_31_n125_m12	26,6101	55,86	2,49E-01
GKD-b_32_n125_m12	35,4893	47,06	2,06E-01
GKD-b_33_n125_m12	29,5375	37,26	2,06E-01
GKD-b_34_n125_m12	35,8491	45,64	2,07E-01
GKD-b_35_n125_m12	31,4543	42,42	2,09E-01
GKD-b_36_n125_m37	273,7160	43,21	6,42E-01
GKD-b_37_n125_m37	350,1520	43,20	6,89E-01
GKD-b_38_n125_m37	315,6070	40,44	7,02E-01
GKD-b_39_n125_m37	263,9250	36,12	7,16E-01
GKD-b_40_n125_m37	317,0930	43,80	6,62E-01
GKD-b_41_n150_m15	50,7852	54,03	2,87E-01
GKD-b_42_n150_m15	46,3515	42,20	2,73E-01
GKD-b_43_n150_m15	42,0455	36,37	2,79E-01
GKD-b_44_n150_m15	48,6170	46,65	2,72E-01
GKD-b_45_n150_m15	39,9460	30,47	2,64E-01
GKD-b_46_n150_m45	441,7440	48,44	1,02E+00
GKD-b_47_n150_m45	405,8140	43,67	1,05E+00
GKD-b_48_n150_m45	368,2450	38,43	1,01E+00
GKD-b_49_n150_m45	421,9380	46,34	1,07E+00
GKD-b_50_n150_m45	397,4910	37,39	1,01E+00

Media Desv:	**35,16**
Media Tiempo:	**3,28E-01**

EXTRA: Algoritmo Memético (AM-(10, 0.1random)) (Cruce de posición)

Algoritmo Memético (SOME_RANDOM) con cruce de posición			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
GKD-b_1_n25_m2	0,0000	0,00	4,55E-01
GKD-b_2_n25_m2	0,0000	0,00	4,55E-01
GKD-b_3_n25_m2	0,0000	0,00	4,56E-01
GKD-b_4_n25_m2	0,0000	0,00	4,56E-01
GKD-b_5_n25_m2	0,0000	0,00	4,58E-01
GKD-b_6_n25_m7	14	7,09	3,25E-01
GKD-b_7_n25_m7	16,6826	15,49	3,26E-01
GKD-b_8_n25_m7	16,7611	0,00	3,29E-01
GKD-b_9_n25_m7	22,9175	25,52	3,26E-01
GKD-b_10_n25_m7	24,5835	5,36	3,28E-01
GKD-b_11_n50_m5	6,5136	70,43	2,95E-01
GKD-b_12_n50_m5	4,2189	49,73	2,99E-01
GKD-b_13_n50_m5	5,0281	53,02	2,94E-01
GKD-b_14_n50_m5	6,1121	72,79	2,89E-01
GKD-b_15_n50_m5	6,1921	53,92	2,89E-01
GKD-b_16_n50_m15	71,0852	39,87	2,92E-01
GKD-b_17_n50_m15	69,2582	30,54	2,92E-01
GKD-b_18_n50_m15	58,5714	26,25	2,96E-01
GKD-b_19_n50_m15	59,4991	21,99	2,93E-01
GKD-b_20_n50_m15	73,6439	35,21	2,90E-01
GKD-b_21_n100_m10	24,8027	44,23	3,39E-01
GKD-b_22_n100_m10	28,3041	51,72	3,42E-01
GKD-b_23_n100_m10	31,7945	51,74	3,49E-01
GKD-b_24_n100_m10	27,0459	68,05	3,51E-01
GKD-b_25_n100_m10	32,3848	46,89	3,48E-01
GKD-b_26_n100_m30	248,9680	32,23	5,80E-01
GKD-b_27_n100_m30	223,0170	43,01	5,77E-01
GKD-b_28_n100_m30	223,1390	52,33	5,75E-01
GKD-b_29_n100_m30	240,0590	42,74	5,76E-01
GKD-b_30_n100_m30	223,0660	42,85	5,76E-01
GKD-b_31_n125_m12	28,3325	58,55	4,16E-01
GKD-b_32_n125_m12	35,7244	47,41	4,19E-01
GKD-b_33_n125_m12	36,2138	48,83	4,19E-01
GKD-b_34_n125_m12	37,7258	48,34	4,16E-01
GKD-b_35_n125_m12	33,5458	46,01	4,11E-01
GKD-b_36_n125_m37	295,3920	47,38	7,60E-01
GKD-b_37_n125_m37	345,9520	42,51	7,60E-01
GKD-b_38_n125_m37	349,3070	46,19	7,58E-01
GKD-b_39_n125_m37	288,2700	41,52	7,38E-01
GKD-b_40_n125_m37	317,4380	43,87	7,57E-01
GKD-b_41_n150_m15	48,7067	52,07	5,09E-01
GKD-b_42_n150_m15	48,0384	44,23	4,97E-01
GKD-b_43_n150_m15	51,0206	47,56	5,08E-01
GKD-b_44_n150_m15	49,6454	47,76	5,00E-01
GKD-b_45_n150_m15	46,8751	40,75	4,96E-01
GKD-b_46_n150_m45	417,7590	45,48	1,02E+00
GKD-b_47_n150_m45	404,3240	43,46	1,01E+00
GKD-b_48_n150_m45	328,4110	30,96	9,94E-01
GKD-b_49_n150_m45	465,7060	51,38	1,02E+00
GKD-b_50_n150_m45	405,9390	38,70	1,00E+00

Media Desv:	**37,92**
Media Tiempo:	**4,97E-01**

EXTRA: Algoritmo Memético (AM-(10, 0.1best)) (Cruce de posición)

Algoritmo Memético (SOME_BEST) con cruce de posición			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
GKD-b_1_n25_m2	0,0000	0,00	1,55E-01
GKD-b_2_n25_m2	0,0000	0,00	1,57E-01
GKD-b_3_n25_m2	0,0000	0,00	1,55E-01
GKD-b_4_n25_m2	0,0000	0,00	1,55E-01
GKD-b_5_n25_m2	0,0000	0,00	1,57E-01
GKD-b_6_n25_m7	14	8,48	1,04E-01
GKD-b_7_n25_m7	15,4204	8,57	1,04E-01
GKD-b_8_n25_m7	17,7741	5,70	1,04E-01
GKD-b_9_n25_m7	18,2380	6,41	1,04E-01
GKD-b_10_n25_m7	23,8596	2,49	1,05E-01
GKD-b_11_n50_m5	4,3550	55,77	8,66E-02
GKD-b_12_n50_m5	4,8051	55,86	8,66E-02
GKD-b_13_n50_m5	6,1083	61,33	8,80E-02
GKD-b_14_n50_m5	5,3771	69,07	8,81E-02
GKD-b_15_n50_m5	5,1220	44,30	8,87E-02
GKD-b_16_n50_m15	62,8103	31,94	1,37E-01
GKD-b_17_n50_m15	58,6926	18,03	1,41E-01
GKD-b_18_n50_m15	61,7759	30,08	1,32E-01
GKD-b_19_n50_m15	62,2867	25,49	1,38E-01
GKD-b_20_n50_m15	64,1338	25,60	1,35E-01
GKD-b_21_n100_m10	21,2711	34,97	1,49E-01
GKD-b_22_n100_m10	26,3402	48,12	1,53E-01
GKD-b_23_n100_m10	25,5713	39,99	1,45E-01
GKD-b_24_n100_m10	26,5153	67,41	1,44E-01
GKD-b_25_n100_m10	25,9321	33,67	1,45E-01
GKD-b_26_n100_m30	231,6050	27,15	4,09E-01
GKD-b_27_n100_m30	188,4480	32,56	4,15E-01
GKD-b_28_n100_m30	200,4310	46,92	4,33E-01
GKD-b_29_n100_m30	202,3210	32,06	4,09E-01
GKD-b_30_n100_m30	194,4920	34,46	3,91E-01
GKD-b_31_n125_m12	26,8329	56,23	2,31E-01
GKD-b_32_n125_m12	33,8383	44,47	1,96E-01
GKD-b_33_n125_m12	30,7785	39,79	1,94E-01
GKD-b_34_n125_m12	37,2213	47,64	1,94E-01
GKD-b_35_n125_m12	30,7514	41,10	2,02E-01
GKD-b_36_n125_m37	257,6490	39,67	6,24E-01
GKD-b_37_n125_m37	321,1260	38,06	6,70E-01
GKD-b_38_n125_m37	334,4470	43,80	6,56E-01
GKD-b_39_n125_m37	286,1660	41,09	6,39E-01
GKD-b_40_n125_m37	355,5890	49,89	5,92E-01
GKD-b_41_n150_m15	43,6465	46,51	2,70E-01
GKD-b_42_n150_m15	49,4505	45,83	2,59E-01
GKD-b_43_n150_m15	45,4060	41,08	2,67E-01
GKD-b_44_n150_m15	48,9116	46,97	2,61E-01
GKD-b_45_n150_m15	46,1147	39,77	2,45E-01
GKD-b_46_n150_m45	436,7630	47,86	9,49E-01
GKD-b_47_n150_m45	356,6020	35,89	1,01E+00
GKD-b_48_n150_m45	367,1990	38,25	9,18E-01
GKD-b_49_n150_m45	420,9280	46,21	1,00E+00
GKD-b_50_n150_m45	389,7930	36,16	9,66E-01

Media Desv:	**34,25**
Media Tiempo:	**3,11E-01**

Resultados globales por tamaño

Comparativa por tamaños (50)				
Caso	Coste medio obtenido	Tamaño	Desv	Tiempo (s)
RandomSearch	73,2566	50,00	62,93	4,67E-01
Greedy	102,7431	50,00	83,15	6,06E-04
LocalSearchRandom	59,7213	50,00	70,26	1,81E-03
LocalSearchHeuristic	55,0002	50,00	67,74	2,27E-03
AGG-uniforme	53,4428	50,00	66,63	1,16E+00
AGG-posición	55,8044	50,00	67,07	9,70E-01
AGE-uniforme	52,8386	50,00	66,37	1,37E+00
AGE-posición	52,8803	50,00	66,95	1,23E+00
AM-(10, 1.0)	34,2853	50,00	43,34	1,21E-01
AM-(10, 0.1)	37,0679	50,00	48,85	3,17E-01
AM-(10, 0.1mej)	35,8112	50,00	43,6628	1,16E-01
AM-(10, 1.0) (Cruce posición)	33,4634	50,00	41,85	1,15E-01
AM-(10, 0.1) (Cruce posición)	36,0123	50,00	45,37	2,93E-01
AM-(10, 0.1mej) (Cruce posición)	33,5467	50,00	41,7464	1,12E-01

Comparativa por tamaños (100)				
Caso	Coste medio obtenido	Tamaño	Desv	Tiempo (s)
RandomSearch	298,2368	100,00	75,54	9,75E-01
Greedy	238,8688	100,00	73,80	4,73E-03
LocalSearchRandom	197,2019	100,00	64,80	1,23E-02
LocalSearchHeuristic	132,5027	100,00	51,67	1,85E-02
AGG-uniforme	170,0324	100,00	58,4429	2,05E+00
AGG-posición	173,6462	100,00	58,7333	1,66E+00
AGE-uniforme	182,4672	100,00	61,7931	2,38E+00
AGE-posición	180,1110	100,00	60,0538	2,08E+00
AM-(10, 1.0)	121,7126	100,00	43,4877	3,11E-01
AM-(10, 0.1)	129,7817	100,00	47,4096	5,05E-01
AM-(10, 0.1mej)	118,7303	100,00	41,3252	2,88E-01
AM-(10, 1.0) (Cruce posición)	120,9660	100,00	42,8049	2,90E-01
AM-(10, 0.1) (Cruce posición)	130,2581	100,00	47,5787	4,61E-01
AM-(10, 0.1mej) (Cruce posición)	114,2927	100,00	39,7315	2,79E-01

Comparativa por tamaños (150)				
Caso	Coste medio obtenido	Tamaño	Desv	Tiempo (s)
RandomSearch	545,8240	150,00	78,65	1,59E+00
Greedy	338,1749	150,00	67,68	1,73E-02
LocalSearchRandom	269,4499	150,00	56,01	4,63E-02
LocalSearchHeuristic	203,2195	150,00	44,96	5,76E-02
AGG-uniforme	296,3334	150,00	57,2778	2,79E+00
AGG-posición	294,2809	150,00	57,4666	2,50E+00
AGE-uniforme	302,6341	150,00	58,0574	3,48E+00
AGE-posición	304,0531	150,00	57,9487	3,05E+00
AM-(10, 1.0)	232,0616	150,00	44,7727	6,90E-01
AM-(10, 0.1)	237,4183	150,00	46,8558	8,01E-01
AM-(10, 0.1mej)	225,0341	150,00	43,0305	6,32E-01
AM-(10, 1.0) (Cruce posición)	226,2977	150,00	42,3998	6,55E-01
AM-(10, 0.1) (Cruce posición)	226,6425	150,00	44,2352	7,55E-01
AM-(10, 0.1mej) (Cruce posición)	220,4814	150,00	42,4530	6,15E-01

Resultados globales totales

Comparativa global por algoritmos			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
RandomSearch	268,2343	60,96	9,16E-01
Greedy	201,8822	66,25	6,37E-03
LocalSearchRandom	155,3244	55,81	1,78E-02
LocalSearchHeuristic	115,7956	47,99	2,28E-02
AGG-uniforme	154,0828	52,5488	1,83E+00
AGG-posición	154,3081	53,4277	1,57E+00
AGE-uniforme	154,6719	53,6758	2,21E+00
AGE-posición	154,3303	53,3229	1,96E+00
AM-(10, 1.0)	112,9694	36,0199	3,47E-01
AM-(10, 0.1)	117,7462	39,2969	5,34E-01
AM-(10, 0.1mej)	111,1092	34,8240	3,20E-01
AM-(10, 1.0) (Cruce posición)	111,6106	35,1638	3,28E-01
AM-(10, 0.1) (Cruce posición)	115,8333	37,9185	4,97E-01
AM-(10, 0.1mej) (Cruce posición)	109,7359	34,2541	3,11E-01
Óptimo	66,4709	0,00	?

Análisis de resultados

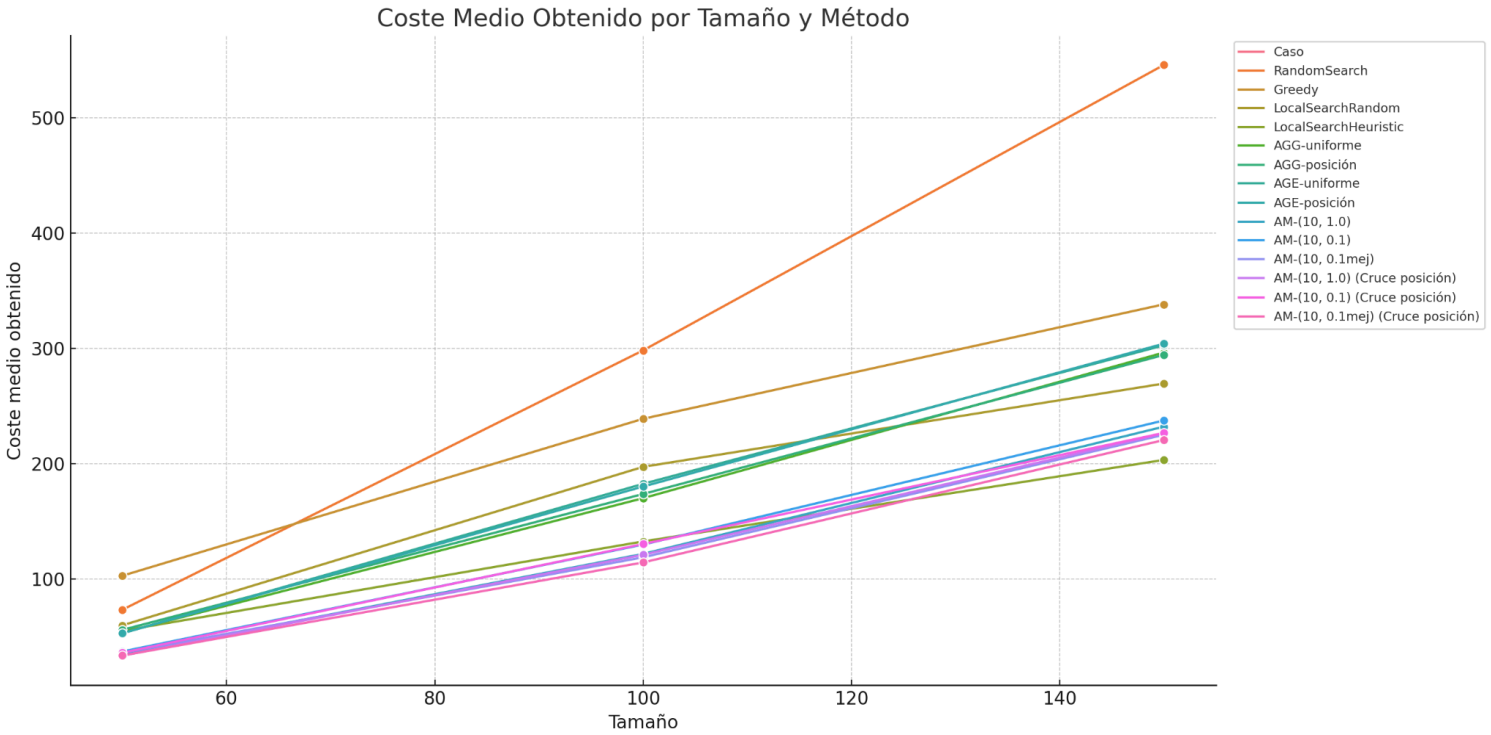
Una vez que tenemos los resultados, vamos a proceder a su análisis. Primero veremos los resultados globales para cada algoritmo, y después haremos un análisis más completo y específico. Solo vamos a analizar a fondo los resultados de las 10 nuevas ejecuciones, los algoritmos de la práctica anterior se incluyen para tener una referencia.

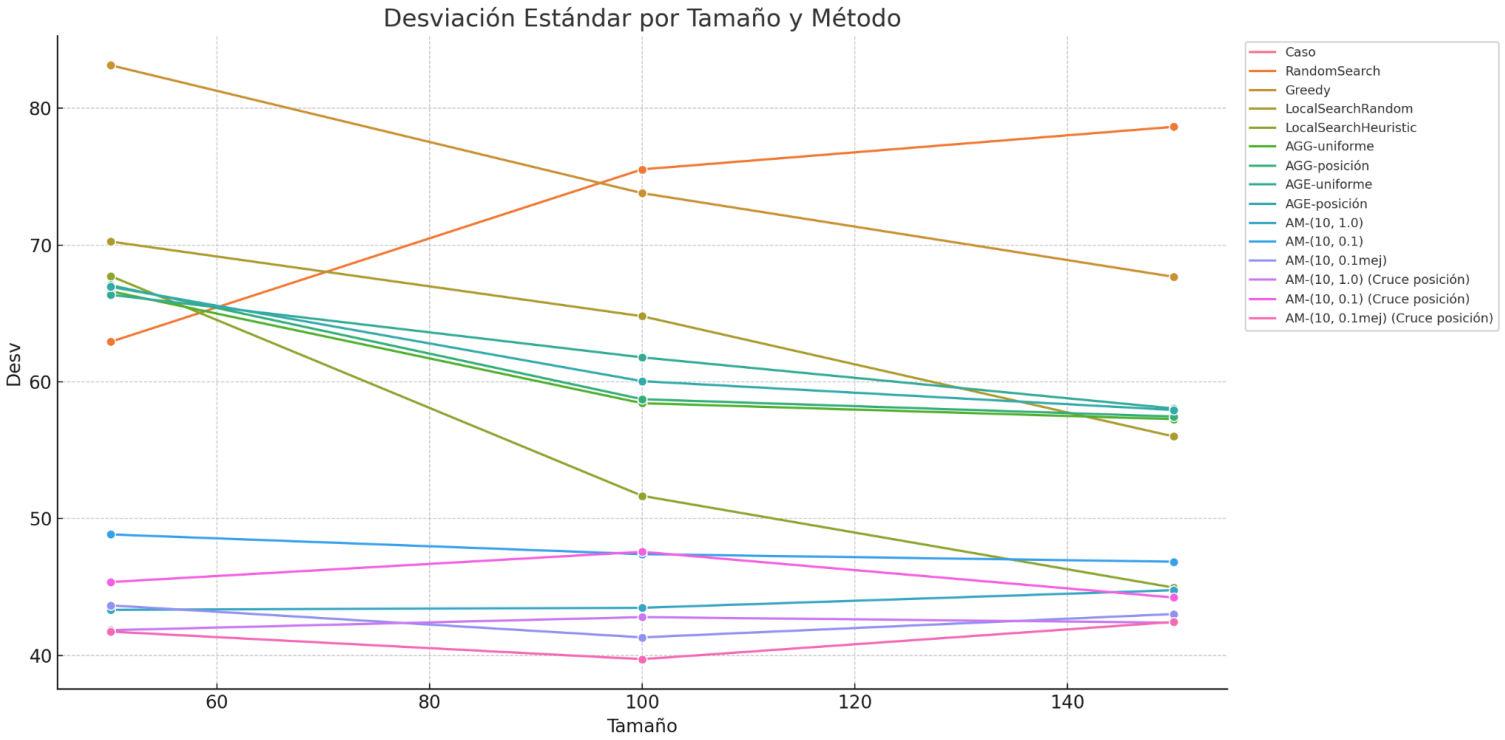
Lo que destaca inmediatamente es que los algoritmos meméticos son una mejora grande respecto a los algoritmos de la práctica anterior. Incluso el peor de ellos (cruce uniforme, 0.1 aleatorio) mejora casi 10 puntos la desviación de la búsqueda local heurística (el mejor algoritmo de la práctica anterior). Entre los seis meméticos, el que obtiene mejores resultados es el AM-(10, 0.1mej) con cruce de posición, con una desviación típica de 34,25 en 0.311 segundos, es el que mejor consigue ajustar el equilibrio entre la exploración del AGG y la explotación de la búsqueda local sobre los mejores candidatos.

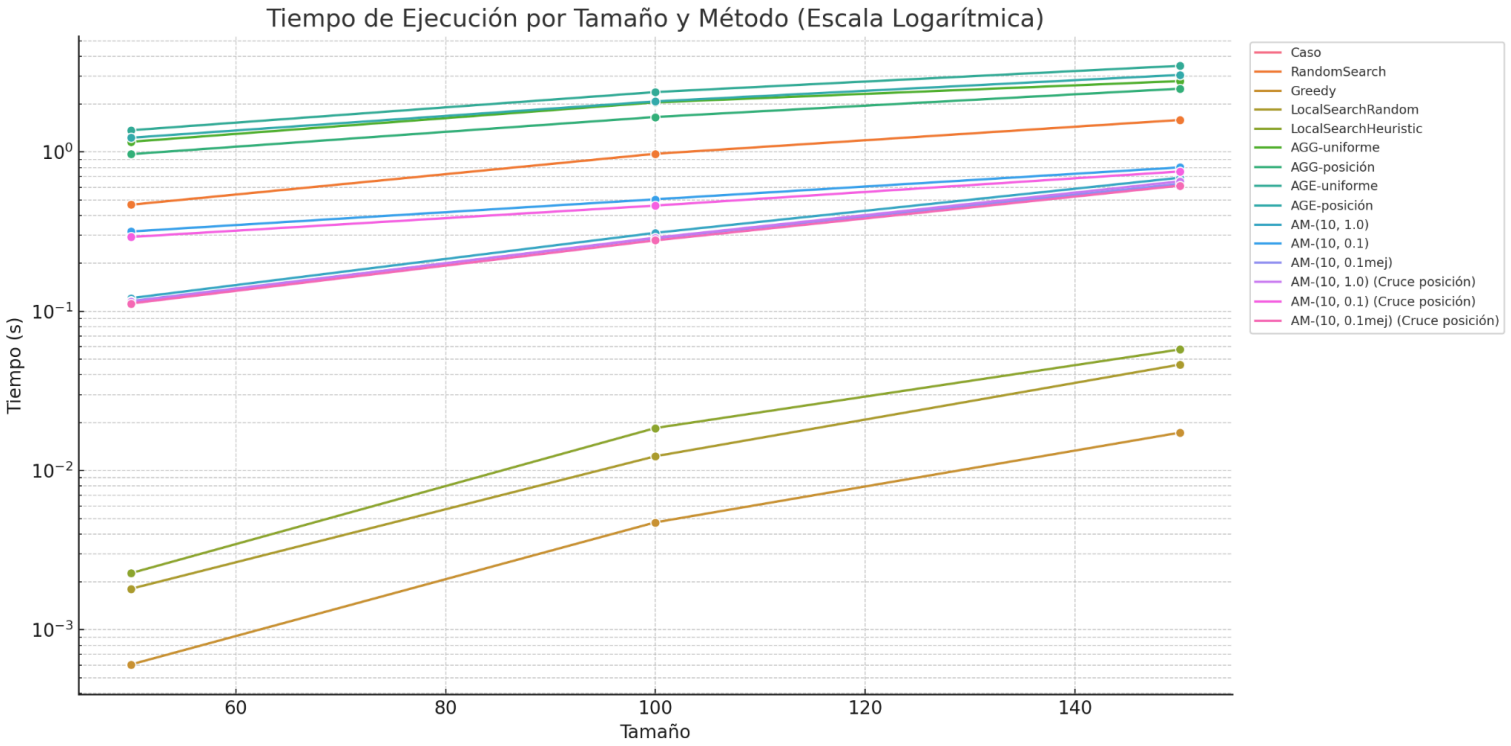
De los algoritmos genéticos, podemos decir que en conjunto son muy lentos, mucho más que cualquier otro algoritmo (tanto de la práctica 1 como de la práctica 2), tardando más de segundo y medio de media (el más lento de la práctica anterior era el random, y tardaba menos de un segundo de media).
Tanto los AGGs como los AGEs muestran unos resultados casi iguales en coste medio obtenido, independientemente del tipo de cruce utilizado, oscilando en torno a 154, con desviaciones típicas entre 52 y 53. Se diferencian entre sí por el tiempo que han tardado en obtener los resultados: los AGEs son considerablemente más lentos que los AGGs, y en ambos casos usar el cruce uniforme ha llevado a tiempos más altos de ejecución. Aún así, siendo estrictos, el mejor resultado en términos de desviación típica ha sido el AGG con cruce uniforme, por tanto esta es la versión que vamos a usar para generar los meméticos.

Como he explicado antes, he tomado la decisión de crear los meméticos con cruce de posición también por su valor didáctico.

Igual que en la práctica anterior, una vez analizados los resultados globales vamos a analizar, otra vez algoritmo a algoritmo, y comparándolos, los resultados por tamaños.







Vemos una primera diferencia respecto a la práctica anterior: generar 100000 soluciones aleatorias ya no consigue mejores resultados que cualquier otro algoritmo para n = 50, y además el tiempo de ejecución del algoritmo aleatorio sigue siendo de los más altos (solo más rápido que los genéticos), de manera que ya no tenemos excusa para elegirlo frente al resto de algoritmos de esta práctica. Habíamos dicho que no íbamos a analizar extensamente los algoritmos de la práctica anterior, así que pasamos al análisis que nos interesa.

Algoritmos genéticos

Son decepcionantes los resultados de los algoritmos genéticos. No mejoran los resultados de la búsqueda local heurística, conllevan más complejidad de implementación y tardan casi 1000 veces más que los algoritmos de la práctica anterior! Son todos muy parecidos, pero si tuviéramos que concretar, los algoritmos genéticos estacionarios son incluso peores que los generacionales, tardan más y obtienen peores resultados. Asumiendo que estuvieran todos los algoritmos implementados, no hay ningún caso donde sería conveniente escoger los algoritmos genéticos, tanto generacionales como estacionarios: en tamaño 50, mejoran la desviación típica de las búsquedas locales y greedy, pero no la de random, y tardan más del doble de tiempo que este último, y en el resto de tamaños la búsqueda local heurística es superior en todos los aspectos. Por supuesto, no hay ni que hablar de compararlos con los algoritmos meméticos, ya que obtienen resultados infinitamente peores en todos los tamaños y tiempos de ejecución mucho, mucho más altos.

Si tuviéramos que elegir un genético de entre todos, nos tendríamos que quedar con el AGG-uniforme, ya que es el “menos malo”, obtiene menor desviación típica que el resto y tarda menos que los estacionarios. Si nos importara mucho el tiempo de ejecución, y solo

pudiéramos elegir entre estos cuatro algoritmos, entonces elegiríamos el AGG-posición, ya que obtiene una desviación muy parecida a la de AGG-uniforme, pero tarda menos. Insisto, esto es solo en caso de tener que elegir uno obligatoriamente, ya que los algoritmos genéticos para este problema no son nada recomendables, No quiere decir que para otros problemas no puedan funcionar, tanto AGGs como AGEs dan buenos resultados en problemas combinatorios, pero en nuestro caso hay opciones de mayor calidad y menor tiempo de ejecución.

Algoritmos meméticos

Pasando a los algoritmos meméticos, los resultados obtenidos son, en líneas generales, excelentes. No solo superan con claridad a los algoritmos genéticos, sino que en casi todos los casos consiguen mejorar los resultados obtenidos por la búsqueda local heurística, que hasta ahora había sido el mejor algoritmo. Incluso consiguen alguna solución óptima con 5 semillas distintas! Fijándonos en los resultados por tamaños, vemos que los algoritmos meméticos son estables en desviación típica variando el tamaño del problema, que nos interesa tanto para tamaños pequeños como para tamaños grandes. Esto contrasta con el comportamiento de los algoritmos de la práctica anterior, concretamente con las búsquedas locales, que mejoraban su calidad cuanto mayor fuera el tamaño del problema.

Comparando el tipo de cruce, esto es lo sorprendente: el cruce de posición obtiene mejores resultados que el cruce uniforme en todos los tamaños de problema! Digo sorprendente porque ya sabemos que los meméticos son esencialmente un AGG a los que se aplica una búsqueda local, y si los resultados obtenidos por el AGG son mejores con el cruce uniforme, sería de esperar que en los meméticos pasara igual. El motivo puede ser que como la heurística usada en el cruce uniforme es casi aleatoria, interfiere con el estilo de trabajo de la búsqueda local, que mejora la solución bit a bit para que luego en el cruce se vuelva a cambiar parte de la solución. El cruce de posición no altera tanto las soluciones ya que no tiene la componente aleatoria para la reparación (no necesita reparación).

Si nos fijamos en las configuraciones de hibridación, vemos que el orden de mejor a menor es el siguiente: cambiar el 10% de los mejores de la población (SOME_BEST), luego cambiar la población completa (ALL) y finalmente cambiar el 10% de la población aleatoriamente (SOME_RANDOM). Entre los dos primeros no hay diferencias grandes, ambos funcionan muy bien y en tiempos competentes, al contrario que SOME_RANDOM, que sí produce resultados algo peores con mayor tiempo de ejecución. Tiene sentido, de la primera práctica hemos aprendido que la aleatoriedad lleva de la mano un tiempo de ejecución mayor, y aquí se ve reflejado a la perfección.

Hablando un poco más de los tiempos de ejecución, los meméticos son por supuesto menos rápidos que las búsquedas locales y greedy, pero no podemos decir que sean tiempos “lentos”, como son los algoritmos genéticos, y además la calidad de los resultados merecen la pena completamente.

Como conclusión, si tuviéramos que elegir un único algoritmo para la resolución de este problema, sería el Algoritmo Memético (AM-(10, 0.1best)) con cruce de posición. Es el que mejores resultados obtiene para cualquier tamaño de problema de entre los estudiados, y el que menos tarda de entre los meméticos. Los algoritmos genéticos no los podríamos

considerar para ningún caso, ya que tienen un coste computacional alto, tardan mucho y los resultados, aunque no podemos decir que sean “malos” como lo eran para el algoritmo greedy de la práctica 1, dejan mucho que desear. El único valor que tienen es ser un paso intermedio entre las búsquedas locales y los algoritmos meméticos, puramente didáctico para este problema. Sobre los algoritmos de la práctica anterior, seguimos pensando lo mismo: si tenemos una restricción de tiempo de ejecución muy limitado, podríamos escoger uno de entre las búsquedas locales o si el tamaño del problema es muy grande, greedy incluso, pero en cualquier otro caso, seguimos quedándonos con los meméticos.

Bibliografía

He utilizado sobre todo los recursos que tenemos en PRADO. También he utilizado inteligencias artificiales generativas para tareas costosas de hacer a mano, como el paso de .txt a .csv, generar las gráficas con los datos, dudas sintácticas de C++ o cómo adaptar el archivo CMakeLists.txt para incluir un archivo ejecutable más. Por último, he buscado más información sobre el problema en información adicional, y sobre los distintos tipos de cruce y AGEs además de información sobre distintos tipos de cruce y SGAs en el siguiente paper: Uniform Crossover in Genetic Algorithms. Trabaja principalmente con GGAs, explorando distintos problemas, y finalmente tiene un apéndice muy útil donde habla de los SGAs (Steady State Genetic Algorithms, AGEs).