Práctica 3

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 seis algoritmos para esta nueva resolución del problema MDD, o al menos, seis variantes de cuatro algoritmos principales: una única versión de BMB (Búsqueda Multiarranque Básica, o Basic Multi-Start en inglés), una única versión del ES (Enfriamiento Simulado, o Simulated Annealing en inglés), dos versiones de ILS (Búsqueda Local Reiterada, o Iterated Local Search en inglés), una con ES y la otra sin ES; y dos versiones de GRASP (Greedy Randomize Adaptative Search Procedure), una con BL y la otra sin BL. En la base, todos comparten la misma estructura común heredada de la clase *MH*, de la que ya se habló en el primer guión de prácticas, de manera que no se repetirá la explicación. La representación de soluciones y la función objetivo son la misma.

Más concretamente, las clases donde se implementan los algoritmos de BMB y GRASP heredan de *MH* directamente, mientras que las clases donde se implementan los algoritmos de ES y ILS sí heredan de *MHTrayectory*, que ya explicamos en la última práctica qué estructura tenía y qué diferencias había con la clase *MH*. Por supuesto, igual que en la última práctica, mantenemos que la clase de la búsqueda local aleatoria herede de *MHTrayectory*, nos ayuda a poder aplicarla a partir de una solución dada, en vez de generar una aleatoria.

Las versiones a utilizar de ILS y GRASP se determinan con un booleano, *useSA* y *useLS* respectivamente. Como sus nombres indican, si están a *true* se usa el enfriamiento simulado en ILS, y la búsqueda local en GRASP, y si están a *false* se usa la búsqueda local en ILS, y nada especial en GRASP.

Como no hay nada más en común a las seis variantes de los algoritmos que hemos implementado, y tampoco nada nuevo respecto al problema, pasamos ya a explicar la estructura de cada algoritmo.

Estructura algoritmos

Basic Multi-Start

El algoritmo de Búsqueda Multiarranque Básica es una mejoría simple respecto a la búsqueda local aleatoria, simplemente genera un número de soluciones aleatorias y les aplica la búsqueda local a todas ellas, devolviendo como resultado la mejor. Para nuestro problema, se hace con diez soluciones iniciales aleatorias, y se le aplica la búsqueda local a cada con un máximo de evaluaciones de 10000, en vez de una única búsqueda local con 100000 evaluaciones.

En términos de implementación, tenemos la clase *BasicMultiStart*, que hereda de *MH*, que cuenta con el método *optimize* cuya implementación es la siguiente, en pseudocódigo:

```
size <- 10
evals <- 0
solutions es un vector vacío
fitness es un vector vacío

para cada i desde 0 hasta size-1 hacer:
        sol <- problem->createSolution()
        solutions[i] <- sol
        fitness[i] <- problem->fitness(sol)
        evals <- evals+1
end para cada

bl <- RandomLocalSearch()
best <- solutions[0]
best_fitness <- problem->fitness[0]
para cada i desde 0 hasta size-1 hacer:
        result <- bl.optimize(problem, solutions[i], fitness[i], maxevals)
        si result.fitness es mejor que best_fitness hacer:
                best <- result.solution
                best_fitness <- result.fitness
        end si
        evals <- evals + result.evaluations
end para cada
devolver {best, best_fitness, evals}
```

No tiene más complicación que crear un vector de soluciones aleatorias y aplicarle a todas la búsqueda local, almacenando la mejor solución en cada iteración. Es una mejora clara a la búsqueda local, pero tiene el inconveniente de que todas las soluciones de las que partimos son aleatorias.

Iterated Local Search

El algoritmo de búsqueda local iterada es una mejora respecto a la búsqueda local aleatoria, y también respecto al algoritmo de búsqueda multiarranque básica. Vimos que el problema del primero era que solía estancarse en óptimos locales, mientras que el problema del segundo era que las soluciones de las que partíamos eran aleatorias.

La búsqueda local iterada supera estos dos problemas introduciendo mutaciones: partiendo de una solución aleatoria, le aplicamos una búsqueda local. Se aplica una mutación sobre la mejor solución hasta el momento y se aplica otra vez la búsqueda local sobre la solución mutada. Se repite esto un número determinado de veces, en nuestro caso, diez (con búsquedas locales de 10000 evaluaciones como máximo).

Como hemos comentado antes, hay dos versiones de este algoritmo: una con búsquedas locales y otra con enfriamientos simulados. Son completamente intercambiables, más adelante se explicará con detalle cómo funciona el ES.

Pasando al operador de mutación, es similar al usado en los algoritmos genéticos y meméticos, pero con algunos cambios. Buscamos una mutación más agresiva, un 20% de la solución se debe cambiar, con mínimo dos cambios por solución. Este es el pseudocódigo de la función *mutate*, que recibe como parámetros la solución a mutar, el porcentaje de mutación y el tamaño del problema:

```
mutated <- original_sol
m <- mutated.size()
num_changes  <- max(m*mut_rate, 2)

para cada i desde 0 hasta num_changes hacer:
        pos <- random(0, m-1)
        // generar un nuevo nodo que no esté en la solución
        hacer:
                new_node <- random(0, n-1)
                exists <- false
                para cada j desde 0 hasta m && not exists hacer:
                        si mutated[j] es igual a new_node hacer
                                exists <- true
                        end si
                end para cada
        mientras exists sea true

        mutated[pos] <- new_node
end para cada

devolver mutated
```

Como se puede apreciar, la única complejidad que tiene el operador de mutación es comprobar que el nodo generado no está en la solución. Una vez que tenemos nuestro operador de mutación implementado, podemos pasar al pseudocódigo del método principal, *optimize*:

```
// conseguir el tamaño del problema sin hacer un cast, fallo que tuve en prácticas anteriores
n <- getSolutionDomainRange().second + 1
bl <- RandomLocalSearch()
sa <- SimulatedAnnealing()
evals <- 0
useSA <- getUseSA()

sol <- current // heredamos de MHTrayectory, nos pasan estos parámetros
fitness <- fit

result <- useSA ? sa.optimize(problem, sol, fitness, maxevals) : bl.optimize(problem, sol, fitness, maxevals) // dependiendo de cómo hemos inicializado la clase, se usa sa o bl
sol <- result.solution
fitness <- result.fitness
evals <- evals + result.evaluations

best <- sol
best_fitness <- fitness

para cada i desde 0 hasta 9 hacer:
        mutated <- mutate(best, 0.2, n)
        mutated_fitness <- problem->fitness(mutated)
        evals <- evals+1

        r <- useSA ? sa.optimize(problem, mutated, mutated_fitness, maxevals) : bl.optimize(problem, mutated, mutated_fitness, maxevals)
        evals <- evals + r.evaluations
        si r.fitness es menor que best_fitness hacer:
                best <- r.solution
                best_fitness <- r.fitness
        end si
end para cada

devolver {best, best_fitness, evals}
```

Igualmente, no tiene mucha complejidad. *getUseSA()* es el *getter* que hemos implementado para la clase *IteratedLocalSearch*, que nos devuelve el atributo *useSA*. Partimos de una primera búsqueda local o enfriamiento simulado sobre una solución aleatoria, y ya se procede como hemos explicado.

Lo bueno de la implementación es poder reutilizar todo el código para las dos variantes, con búsqueda local o enfriamiento simulado. Solo cambiando la inicialización del algoritmo, pasándole un *true* o un *false* al constructor, se tiene.

Enfriamiento simulado

El algoritmo de enfriamiento simulado está inspirado en el proceso físico de enfriamiento de metales. Parte de una solución inicial y explora el vecindario (mismo concepto de vecindario que para la búsqueda local, ya se explicó en detalle en la primera práctica) aceptando todas las soluciones que mejoren la función objetivo, y con cierta probabilidad, también las soluciones que sean peores que la actual, lo que permite escapar de óptimos locales. Esta probabilidad depende de la temperatura, un parámetro que va disminuyendo durante la ejecución, de manera que aceptamos menos soluciones malas a medida que avanza el algoritmo.

La temperatura se actualiza en cada iteración según el esquema de enfriamiento de Cauchy modificado, que se ha explicado en clase. Sigue la siguiente fórmula:

$$T_{k+1} = \frac{T_k}{1 + \beta \cdot T_k} \qquad \beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

donde *M* es el número de enfriamientos a realizar, *T0* es la temperatura inicial y *Tf* es la temperatura final, que tendrá un valor cercano a cero. A su vez, la temperatura inicial se calculará en función de la siguiente fórmula:

$$T_0 = \frac{\mu \cdot Coste(S_0)}{-ln(\varphi)}$$

donde *Coste(S0)* es el coste de una solución inicial *S0* y $\varphi \in [0, 1]$ es la probabilidad de aceptar una solución un μ por 1 peor que la inicial. Los valores de estos parámetros vienen dados en el guión de la práctica.

Pasando a la implementación, usamos un método auxiliar para calcular el vecindario de la solución *current*. Este es su pseudocódigo:

```
m <- current.size()
neighbourhood es un vector de pares de enteros

para cada i desde 0 hasta n-1 hacer:
        si i no está en current hacer:
                available.push_back(i)
        fin si
fin para cada

para cada i desde 0 hasta m hacer:
        para cada j en available hacer:
                neighbourhood.emplace_back(i, j)
        fin para cada
fin para cada

neighbourhood <- shuffle(neighbourhood)
devolver neighbourhood
```

Es la misma idea que se siguió en la búsqueda local de la primera práctica, encapsulada en un método de la clase *SimulatedAnnealing.* Ahora sí, podemos pasar a explicar el pseudocódigo de la clase *optimize*.

```
// valores de los parámetros
mu <- 0.2
phi <- 0.3
Tf <- 0.001
m <- problem.getSolutionSize()
n <- problem.getSolutionDomainRange().second+1
max_neighbours <- 100 * m
max_improvements <- 0.1 * max_neighbours
max_iters <- maxevals / max_neighbours

sol <- current // heredamos de MHTrayectory, nos pasan estos parámetros
fitness <- fit
evals <- 1

best <- sol
best_fitness <- fitness

T0 <- (mu*fitness) / (-log(phi))
mientras que T0 sea menor que Tf hacer:
        Tf <- Tf / 10
fin mientras
T <- T0
beta <- (T0 - Tf) / (max_iters * T0 * Tf)

improvement_num <- -1
// bucle exterior
mientras que evals < maxevals && T < Tf && improvement_num != 0 hacer:
        improvement_num <- 0
        neighbour_num <- 0
        neighbourhood <- getNeighbourhood(sol, n)
        neighbourhood_index <- 0

        // bucle interior
        mientras que improvement_num < max_improvements && neighbour_sum < max_neighbours && evals < maxevals && neighbourhood_index <- neighbourhood.size() hacer:
                neighbour <- sol
                neighbour[neighbourhood[neighbour_index].first] <- neighbourhood[neighbour_index].second // creamos el vecino cogiendo el neighbour_index-ésimo elemento del vecindario
                neighbour_fitness <- problem->fitness(neighbour)
                evals <- evals+1
                neighbour_num <- neighbour_num+1
                delta <- neighbour_fitness - fitness
```

```
            si delta < 0 || random(0, 1) <= exp(-delta/T) hacer:
                sol <- neigbour
                fitness <- neighbour_fitness
                improvement_num <- improvement_num+1
                // reiniciamos el vecindario
                neighbourhood <- getNeighbourhood(sol, n)
                neighbour_index <- -1

                si fitness < best_fitness hacer:
                    best <- sol
                    best_fitness <- fitness
                end si
            end si
            neighbour_index <- neighbour_index+1
        fin while (bucle interior)
        T <- T / (1 + beta * T) // enfriamiento
fin while (bucle exterior)

devolver {best, best_fitness, evals}
```

De esta implementación sí hay más cosas que comentar. Comenzamos con darle a los parámetros el valor que tienen que tener y definiendo las variables a utilizar, nada nuevo hasta aquí. Sabemos que en todo momento tenemos que comprobar que la temperatura nunca sea menor que la temperatura final, y eso empieza por comprobar que la inicial no lo sea (el bucle está para asegurarnos).

El resto de la implementación es bastante estándar, es muy similar a la de la búsqueda local pero aceptando algunas soluciones malas (para eso está la segunda comprobación del delta. Para asegurar que no se repiten vecinos hemos creado un vecindario (en vez de ir creando vecinos uno a uno), por lo cual hemos tenido que añadir una comprobación extra en el bucle interno, y es que el vecindario no se quede vacío (en caso de que pase, terminamos). No he utilizado el fitness factorizado (siendo consciente de que había que hacerlo) por falta de tiempo, no lo implementé con la API en su momento en la práctica 1 y ese error me ha lastrado hasta ahora. De todas maneras, he comprobado resultados con otros compañeros que sí lo han utilizado, y los tiempos de ejecución son similares, de manera que no es algo especialmente grave en este caso.

GRASP

Pasamos al último algoritmo. El algoritmo GRASP está basado en la construcción iterativa de soluciones de forma heurística. En cada iteración, se construye una solución utilizando un *greedy* modificado (usando una lista restringida de candidatos, *LRC*), generada a partir del criterio heurístico visto en los apuntes:

$$heur_{min} + \alpha \cdot (heur_{max} - heur_{min}), \text{ aplicando un } \alpha = 0{,}2.$$

y se entrega la mejor. Como hemos explicado, vamos a implementar dos variantes: una entregará la mejor solución aleatoria sin aplicar una búsqueda local, y la otra sí aplicará una búsqueda local a cada solución aleatoria (igual que con BMB).

Pasando a la implementación, hemos usado dos métodos auxiliares que se usan en el método principal del algoritmo, *optimize*. El primero es el método *getDisp*, que nos devuelve la dispersión calculando la distancia de los elementos candidatos a los actualmente seleccionados. Su psuedocódigo es:

```
min_d <- 100000
max_d <- -1

para cada s en selected hacer:
        d < -matrix[s][candidate]
        min_d <- min(min_d, d)
        max_d <- max(max_d, d)
end para cada

devolver max_d - min_d
```

Este método es llamado en el segundo método auxiliar, *getGreedySolution*, que como su nombre indica, nos construye una solución *greedy* aleatorizada de la manera que hemos explicado. Su pseudocódigo es el siguiente:

```
available es un vector de tamaño n
para cada i desde 0 hasta n-1 hacer:
        available[i] <- i
fin para cada

e0 <- random(0, n-1)
hacer:
        e1 <- random(0, n-1)
mientras e1 == e0 // sin repetición

selected.push_back(e0)
selected.push_back(e1)
borrar e0 de available
borrar e1 de available
```

```
// bucle principal
mientras selected.size() < m hacer:
        lc y heur son vectores vacíos
        para cada i desde 0 a n-1 hacer:
                si i está en selected hacer:
                        disp <- getDisp(selected, i, matrix)
                        lc.push_back(i)
                        heur.push_back(disp)
                fin si
        fin para cada

        hmin <- mínimo de heur
        hmax <- máximo de heur
        mu <- hmin + 0.2*(hmax-hmin)

        lrc es un vector vacío
        para cada i desde 0 hasta lc.size()-1 hacer:
                si heur[i] <= mu hacer:
                        lrc.push_back(lc[i])
                fin si
        fin para cada

        chosen <- lrc[random(0, lrc.size()-1)]
        selected.push_back(chosen)
        borrar chosen de available
fin mientras // ya tenemos la solución completa

devolver selected
```

Es una manera muy parecida de construir la solución a la que hicimos para el algoritmo *greedy* de la práctica 1. La diferencia clave es que en vez de escoger directamente el elemento que menos dispersión aporta, se escoge uno aleatorio de entre los que tienen una dispersión menor o igual que *mu.* Podemos pasar finalmente al pseudocódigo del método *optimize*:

```
// cast permitido para conseguir la matriz de distancias
mdd_problem <- dinamic_cast<MDDProblem*>(problem)
m <- problem->getSolutionSize()
n <- problem->getSolutionDomainRange().second+1
matrix <- mdd_problem->getMatrix()

bl <- RandomLocalSearch()
evals <- 0
useLS <- getUseLS()
best es una tSolution sin inicializar
best_fitness es un tFitness con valor máximo
```

```
para cada i desde 0 hasta 9 hacer:
        candidate <- getGreedySolution(m, n, matrix)
        fitness <- problem -> fitness(candidate)
        evals <- evals+1

        si useLS es true hacer:
                result <- bl.optimize(problem, candidate, fitness, maxevals)
                candidate <- result.solution
                fitness <- result.fitness
                evals <- evals + result.evaluations
        fin si

        si fitness < best_fitness hacer:
                best < candidate
                best_fitness <- fitness
        fin si
fin para cada

devolver {best, best_fitness, evals}
```

Aquí, seguimos un modus operandi muy parecido al de BMB: se repite 10 veces el mismo proceso y cogemos el mejor resultado. Como en ILS, la diferencia de implementación entre usar la búsqueda local y no usarla es es simplemente un booleano que recibe el constructor de la clase como parámetro. Las máximas evaluaciones para cada búsqueda local son 10000, que es el parámetro que le pasamos directamente al método *optimize* para redirigirlo a la llamada del método *optimize* de la búsqueda local.

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 2.**

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:

- *mddp.h*
- *greedy.h*
- *randomLS.h*
- *GA.h*
- *GGA.h*
- *MA.h*
- ***bmb.h***
- ***ils.h***
- ***simulatedAnnealing.h***
- ***grasp.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:

- *mddp.cpp*
- *greedy.cpp*
- *randomLS.cpp*
- *GA.cpp*
- *GGA.cpp*
- *MA.cpp*
- ***bmb.cpp***
- ***ils.cpp***
- ***simulatedAnnealing.cpp***
- ***grasp.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. **De prácticas anteriores**, *GGA.cpp* y *randomLS.cpp* contienen el *optimize* respectivo para cada algoritmo.

En relación a esta nueva práctica, *bmb.cpp* incluye la implementación del método principal *optimize* para el algoritmo bmb, *grasp.cpp* incluye la implementación de los métodos auxiliares que hemos explicado en la sección correspondiente y del método principal *optimize* para el algoritmo grasp, *simulatedAnnealing.cpp* igual para el algoritmo de enfriamiento simulado, y *ils.cpp* igual para la búsqueda local iterada. Por cuestiones de implementación de la clase del algoritmo memético, incluimos los archivos GA.h, GGA.h y GA.cpp, GGA.cpp (heredan de esas clases).

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 **6 variaciones de los 4 algoritmos** con 5 semillas distintas, para todos los archivos de datos del directorio *data*, junto con el tiempo que tarda cada uno **y las ejecuciones medias, además de un resumen por tamaños, para facilitar esa sección del 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 (*BMB, ILS-BL, ILS-ES, ES, GRASP-NOBL, GRASP-SIBL)* 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.

Resultados

Resultados globales por tamaño

Comparativa por tamaños (50)					
Caso	**Coste medio obtenido**	**Tamaño**	**Desv**	**Tiempo (s)**	**Evaluaciones**
RandomSearch	73,2566	50,00	62,9292	4,67E-01	100000
Greedy	102,7431	50,00	83,1464	6,93E-04	1
LocalSearchRandom	59,7213	50,00	70,2575	1,16E-03	935,24
BMB	38,8551	50,00	53,3006	1,22E-02	9532,88
ILS	37,8551	50,00	50,8506	1,06E-02	8862,74
ILS-ES	36,3981	50,00	48,8819	1,03E-01	15322
SImulatedAnnealing	56,4217	50,00	67,7459	1,00E-02	1558,56
GRASP-NOBL	158,1813	50,00	88,6434	1,76E-03	10
GRASP-SIBL	38,1340	50,00	52,4613	1,35E-02	9554,8
AM-(10, 0.1mej) (Cruce posición)	33,5467	50,00	41,7464	1,15E-01	100000

Comparativa por tamaños (100)					
Caso	**Coste medio obtenido**	**Tamaño**	**Desv**	**Tiempo (s)**	**Evaluaciones**
RandomSearch	298,2368	100,00	75,5369	9,75E-01	100000
Greedy	238,8688	100,00	73,8019	4,39E-03	1
LocalSearchRandom	197,2019	100,00	64,8005	9,42E-03	4092,96
BMB	122,4341	100,00	43,5888	1,05E-01	44326,3
ILS	119,5653	100,00	43,0336	8,34E-02	41337,6
ILS-ES	124,3588	100,00	44,1631	7,49E-01	45379,4
SimulatedAnnealing	171,0927	100,00	58,3783	8,22E-02	5882,72
GRASP-NOBL	514,5308	100,00	87,8732	8,56E-03	10
GRASP-SIBL	124,3974	100,00	43,0887	1,03E-01	43099
AM-(10, 0.1mej) (Cruce posición)	114,2927	100,00	39,7315	2,86E-01	100000

Comparativa por tamaños (150)					
Caso	Coste medio	Tamaño	Desv	Tiempo (s)	Evaluaciones
RandomSearch	545,8240	150,00	78,6517	1,59E+00	100000
Greedy	338,1749	150,00	67,6848	1,73E-02	1
LocalSearchRandom	269,4499	150,00	56,0095	3,57E-02	11321,4
BMB	224,7278	150,00	40,5041	2,93E-01	75025,9
ILS	215,6121	150,00	39,8542	2,36E-01	73470,7
ILS-ES	250,0396	150,00	46,0972	1,94E+00	51528,9
SImulatedAnnealing	272,6802	150,00	57,0894	2,93E-01	12166,2
GRASP-NOBL	820,3305	150,00	86,7949	2,34E-02	10
GRASP-SIBL	235,0890	150,00	44,3629	2,98E-01	73969,5
AM-(10, 0.1mej) (Cruce posición)	220,4814	150,00	42,4530	6,34E-01	100000

Resultados globales totales

Comparativa global por algoritmos				
Caso	Coste medio	Desv	Tiempo (s)	Evaluaciones
RandomSearch	268,2343	60,9631	9,16E-01	100000
Greedy	201,8822	66,2486	6,37E-03	1
LocalSearchRandom	155,3244	55,8101	1,78E-02	4968,71
BMB	114,3102	38,3812	1,20E-01	38255,7
ILS	109,2105	38,1906	9,59E-02	36818,3
ILS-ES	122,1484	39,2034	7,78E-01	31494,5
SimulatedAnnealing	151,9751	53,9672	1,11E-01	5990,64
GRASP-NOBL	435,1306	77,9532	9,67E-03	10
GRASP-SIBL	115,1040	38,8751	1,21E-01	37989,4
AM-(10, 0.1mej) (Cruce posición)	109,7359	34,2541	3,19E-01	100000
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. Cabe destacar que he vuelto a ejecutar los algoritmos *Greedy* y *RandomLocalSearch* para tener los tiempos de ejecución a la par con los de los algoritmos de esta práctica, y para obtener el número de ejecuciones de *RandomLocalSearch*. Por supuesto, los resultados de coste y desviación de estos dos algoritmos son los mismos que los de la primera práctica.

Tenemos unos resultados interesantes para comentar. Lo primero es reconocer que ninguno de los algoritmos nuevos de esta práctica ha conseguido superar (al menos en desviación típica) los resultados del algoritmo memético de la práctica anterior. Tiene sentido, vimos que los algoritmos basados en trayectorias no aportan una mejora sustancial cuando nuestras búsquedas locales no suelen estancarse en óptimos locales. Pero esto no termina aquí, ya que casi todos los algoritmos que hemos implementado en esta práctica han obtenido resultados muy, muy respetables. Vamos a verlo en profundidad.

Vamos a empezar por el peor algoritmo: la versión de *GRASP* sin búsqueda local. Obtiene resultados catastróficos, mucho peores que random y *greedy* (que estaban establecidos firmemente como los peores hasta ahora). Encima tarda 1.5 veces más que el algoritmo greedy, no he encontrado ninguna posible justificación para usarlo: es más complejo de implementar que el *greedy*, obtiene peores resultados y tarda más en ejecutar. En términos de coste medio, el algoritmo *greedy* se salvaba un poco en tamaños grandes de problemas. A esta versión no se le da bien ni eso, obtiene un coste medio casi tres veces mayor que su predecesor para tamaño 150, y casi dos veces mayor que el algoritmo aleatorio, cuyos resultados ya analizamos en profundidad en la práctica uno. Todo esto cobra sentido si nos fijamos en la implementación del algoritmo: no llegamos a efectuar un algoritmo *greedy* como tal, empezamos igual pero luego escogemos aleatoriamente el nodo a insertar en la solución, algo que añade mucha variedad a la solución (pero no la aprovechamos en esta versión del algoritmo!). A partir de ahora, no vamos a incluir más comparaciones con esta versión del algoritmo *GRASP*, ya que no nos aporta mucho: nos vale con quedarnos con que no merece la pena usarlo para este problema.

La transición natural es pasar a hablar de la versión de *GRASP* que sí usa una búsqueda local. Hemos mencionado que introducir posibles nodos buenos aleatoriamente nos da mucha variedad, pero el problema es que si luego no la aprovechamos, nos quedamos con una solución seguramente peor de la que podríamos haber conseguido si cogiéramos a cada paso la mejor opción (*greedy*). Al incorporar una búsqueda local, aprovechamos el sacrificio hecho a la hora de crear la solución no-óptima y lo explotamos con la búsqueda local, obteniendo, como se puede ver en las tablas, resultados muy competitivos en todos los tamaños. Concretamente, el algoritmo *GRASP-SIBL* es el tercero mejor de los implementados en esta práctica, pero con el top 4 siendo muy, muy ajustado. En términos de tiempo y de número de evaluaciones, está en la media de los cuatro mejores algoritmos de esta práctica, luego comentaremos más sobre esto.

Pasamos al resto de algoritmos que constituyen el top 4 de esta práctica: *ILS*, *BMB* y *ILS-ES*. Empezamos por el primero implementado, y el más simple: *BMB*. Los buenos resultados que alcanza sorprenden por lo simple que es la estrategia: hacer una búsqueda local sobre 10 soluciones aleatorias y quedarnos con la mejor. Su coste medio es consistentemente bueno, su tiempo de ejecución no podemos decir que sea lento (tres veces más rápido que el algoritmo memético), y su desviación se mantiene relativamente estable a lo largo que cambian los tamaños (igual que la búsqueda local, cuanto mayor sea el tamaño del problema mejor funciona). Sabíamos que el principal problema de nuestra búsqueda local eran los óptimos locales, y encontramos en el algoritmo *BMB* una manera simple de superar parcialmente este problema, a cambio de un tiempo de ejecución ligeramente más alto. Le vale para conseguir la medalla de plata de los algoritmos de esta práctica!

Los resultados del algoritmo *ILS* con búsqueda local demuestran que se trata de unas mejores opciones entre las metaheurísticas basadas en trayectorias que conocemos, al menos para este problema. Ya hemos explicado las diferencias con el algoritmo *BMB*: al introducir el operador de mutación se explora más agresivamente el espacio de soluciones, y son esas mismas mutaciones agresivas las que le hacen ser el algoritmo que mejor trata los óptimos locales. No tiene nada que envidiarle al algoritmo memético, consiguiendo mejor coste medio global, y mejor coste medio y desviación típica en tamaños grandes (150).

Además, es considerablemente más rápido y conlleva menos evaluaciones, de manera que, personalmente, lo considero el ganador incluso por encima del algoritmo memético, ya que produce soluciones igual de buenas (o casi) en tiempos mucho más competitivos que el memético. Es curioso ver que aunque tenga un coste medio global menor que el del algoritmo memético (109,2105 frente a 109,7359) la desviación es cuatro puntos mayor. Esto se debe a que en tamaños pequeños, las pequeñas diferencias de coste tienen un mayor impacto en la desviación.

Nos quedan dos algoritmos de los que hablar: el de *Enfriamiento Simulado,* y la versión de *ILS* que lo utiliza. Toca hablar del papel que ha desempeñado el primero en esta práctica. Es una alternativa a la búsqueda local que obtiene mejores resultados, otra vez teniendo que ver con la manera de evitar los óptimos locales, como hemos visto en el resto de algoritmos. Aún así, no está a la altura de *ILS* o *BMB* por lo que hemos comentado al principio del análisis, *MDD* no es un problema donde los óptimos locales afecten especialmente. Obtiene resultados aceptables en un tiempo medianamente aceptable, tardando 10 veces más que la búsqueda local. Parece que funciona mejor en tamaños de problema pequeños, como vemos en la tabla, pero no es una gran mejoría y la diferencia de tiempo importa bastante. Esto explica por qué el algoritmo que lo utiliza, *ILS-ES,* tarda tanto, que nos da pie a hablar del último algoritmo de esta práctica.

Los resultados de *ILS-ES* son un poco decepcionantes, a mi parecer. En términos de coste medio y desviación típica, está dentro del top 4 del que hemos hablado (aunque en último lugar), de manera que no se le puede recriminar mucho por ahí, pero lleva un tiempo de ejecución, sobre todo comparado con el resto de algoritmos que constituyen el top. Tarda casi 10 veces más que la versión del mismo algoritmo que usa búsqueda local (concretamente, para los últimos archivos de tamaños n = 150, m = 45 tarda más de 3,5 segundos en ejecutar cada uno de ellos), y encima tiene peor coste medio y desviación. Esto es algo que me ha extrañado personalmente, ya que intuía que si el algoritmo de enfriamiento simulado funcionaba mejor que el algoritmo de búsqueda local aleatoria, la variante de *ILS* que utilizaba el primero también debería haber funcionado mejor que la segunda.
Me recuerda un poco a la práctica anterior, donde el cruce de posición funcionaba mejor para los algoritmos meméticos que el cruce uniforme, cuando en el algoritmo genético generacional (algoritmo en el que se basaba nuestro algoritmo memético) obtenía mejores resultados con el cruce uniforme que con el cruce de posición.
Pensando en un motivo para que suceda esto, nos damos cuenta que puede tener sentido: hemos visto que tarda 10 veces más el enfriamiento simulado que la búsqueda local,y por otra parte, el ES puede introducir más variabilidad en los resultados, algo que no ayuda especialmente a un esquema como el de *ILS* que trata de acumular mejoras consistentes.

Igual que en el resto de las prácticas, es momento de una pequeña conclusión. Podemos decir que los algoritmos basados en trayectorias, y en particular aquellos que combinan bien explotación (BL) y exploración (mutaciones o aceptación de soluciones peores), son la mejor estrategia para este problema.
Si no tenemos restricciones de tiempo ni de recursos, el *AM(10, 0.1mej)* con cruce por posición sigue siendo la mejor opción, sobre todo en tamaños de problema pequeño. En tamaño de problemas grandes sería mejor usar *ILS* directamente, pero la versión que usa búsqueda local en vez de enfriamiento simulado. BMB no es una mala opción en absoluto

visto lo simple que es de implementar y los buenos resultados que genera. En cambio, pienso que GRASP con búsqueda local no aporta nada que no pueda cumplir por sí solo el algoritmo *ILS*. Por supuesto, no consideramos ni *RandomSearch*, ni *GRASP* sin búsqueda local para ningún caso, y *greedy* solo si tuviéramos un tiempo de ejecución muy, muy reducido en problemas grandes (como ya comentamos en la práctica 1).

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. Para esta práctica, no he encontrado ningún *paper* cuyo contenido nos fuera relevante.