Práctica 1

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 tres algoritmos para la resolución del problema MDD. Todos ellos comparten una estructura común, heredada de la clase MH, que únicamente incluye un constructor y un destructor por defecto y el método *optimize*, el cual recibe como parámetros el problema que van a estudiar y el número máximo de evaluaciones antes de parar la ejecución del algoritmo. Este es el método que se llamará desde el *main* para estudiar cómo se comporta cada algoritmo a la hora de resolver el problema.

Pero antes de eso, vamos a hablar de cómo se representan las soluciones del problema, y para ello necesitamos hablar de cómo hemos representado nuestro problema. La clase *MDDProblem*, heredada de la clase *Problem*, tiene como atributos privados *n*, el número de nodos que constituyen nuestro problema, *m*, el tamaño que tendrá nuestra solución, y *matriz*, una matriz de datos de punto flotante que, como hemos comentado antes, incluye las distancias entre los nodos *i* y *j*. Hemos implementado un constructor que pueda inicializar estos atributos a partir de un fichero de texto, que será uno de los casos de la librería MDPLIB. También cuenta con los getters de *n*, *m* y *matriz*.

Una vez que tenemos una instancia de nuestro problema definida, una solución candidata viene dada por *tSolution*, el tipo de dato que usamos que, en nuestro caso, es un vector de enteros. Como hemos visto en la descripción del problema, cada una de las *m* posiciones del vector es el índice del nodo que forma parte de dicha solución.

Otro método que tiene nuestra clase *MDDProblem* es *CreateSolution*, la cual devuelve una solución aleatoria al problema. Hace lo siguiente:

```
solucion vector vacío
indices vector vacío
para todo i desde 0 hasta n - 1 hacer:
        indices <- i
desordenar indices
para todo i desde 0 hasta m - 1 hacer:
        solucion <- indices(i)
devolver solucion
```

Es muy simple, creamos un vector de índices del 0 al *n-1,* lo desordenamos, y cogemos los *m* primeros. Una vez que tenemos una solución, tenemos que ser capaces de calcular cómo de buena es. Esto lo hacemos mediante el método *fitness*, también común a los tres algoritmos, y cuyo pseudocódigo es el siguiente:

```
sumas vector vacío
s es la longitud de solucion
para todo i desde 0 hasta s - 1 hacer:
        suma <- 0
        para todo j desde 0 hasta s - 1 hacer:
                suma <- distancia entre solucion(i) y solucion(j)
        añadir suma a sumas
devolver max(sumas) - min(sumas)
```

Lo particular de nuestra implementación de la función objetivo es que puede evaluar soluciones parciales. Esto es, no es necesario que una solución tenga longitud *m* para evaluar lo buena que es. El único cambio que hay que hacer es que *s* sea la longitud de *solucion* (la cual recibimos como parámetro), en vez de que sea *m* automáticamente. Nos va a ser muy útil a la hora de implementar el algoritmo greedy. Más sobre esto más adelante.

Finalmente, estamos preparados para definir *ResultMH*, un struct que incluye una tSolution, su fitness, calculado mediante la función descrita antes, y el número de evaluaciones que se han llevado a cabo finalmente. La función *optimize* devuelve una instancia del struct *ResultMH*. A partir de esta instancia se exponen los resultados en el main y se hacen los cálculos necesarios para su posterior análisis.

Estructura algoritmos

Algoritmo greedy

Los algoritmos greedy son aquellos que se basan únicamente en la información del estado actual para escoger la siguiente alternativa. Es decir, siempre va a escoger la mejor opción que se puede obtener a partir del estado actual como siguiente paso.

En nuestro problema, eso se traduce en escoger a cada paso el mejor nodo de entre todos los disponibles. Empezamos con un primer nodo aleatorio, y a partir de ahí construimos la solución buscando el siguiente nodo que más disminuye (o menos aumente) la dispersión.

Esa es la heurística que vamos a utilizar: la misma función *fitness* que habíamos adaptado para que admitiera soluciones parciales. Se le pasa como parámetro la solución candidata, que en la primera iteración, incluirá el nodo seleccionado aleatoriamente y el nodo sobre el que estemos iterando. Únicamente en este caso, da igual qué nodo se introduzca finalmente en la solución, ya que cualquier solución de dos nodos va a tener un fitness de 0. Para el resto de casos, se evalúa cómo de buena es, y se devuelve su fitness. Si es mejor que el que teníamos anteriormente, se incluye el nodo en la nueva mejor solución, y se actualiza el mejor fitness. Se repite el proceso iterando sobre los nodos disponibles. No incluimos el pseudocódigo de la función heurística por haberlo escrito en la sección anterior.

Pasamos a analizar la implementación del método optimize. Este es su pseudocódigo:

```
mdd_problem <- problem // cast
n <- tamaño del problema
m <- tamaño de la solución
disponibles es un vector con todos los elementos
seleccion es un vector vacío

e0 elemento aleatorio de disponibles
añadir e0 a seleccion
quitar e0 de disponibles
mientras la solución no esté completa hacer:
        mejor_dispersion es un número muy grande
        mejor_elemento es un índice cualquiera
        para cada elemento e disponible hacer:
                candidata copia de seleccion
                añadir e a candidata
                dispersion <- fitness(candidata)
                si dispersion < mejor_dispersion hacer:
                        mejor_dispersion = dispersion
                        mejor_elemento = e
        añadir mejor_elemento a seleccion
        quitar mejor_elemento de disponibles

final_fitness = fitness(seleccion)
devolver {seleccion, final_fitness, 1)
```

Como hemos comentado, vemos que nuestra implementación va construyendo la mejor solución según la inclusión de un nuevo nodo mejora (o empeora menos) la dispersión de entre todos los disponibles. Por supuesto, cuando se selecciona un nodo para añadirlo a la solución, se elimina de los disponibles. Como habíamos comentado en el apartado anterior, se devuelve un objeto *ResultMH* compuesto por la solución final, su fitness y el número de evaluaciones. En este caso, solo se ejecuta una vez el algoritmo, de manera que este último valor no nos hace falta. Devolvemos 1.

Algoritmos de búsqueda local

Los algoritmos local search (o búsqueda local) son aquellos que, dada una solución inicial, realizan cambios, nodo a nodo, con la intención de mejorar la dispersión de la solución. Para nuestro problema, consideramos el esquema del primero mejor, es decir, cuando realicemos un cambio que mejore la dispersión, pasamos a trabajar sobre esta nueva solución, sin ver si habría otro cambio sobre la solución original que mejorara más todavía la dispersión.

Para saber qué cambios se pueden hacer sobre la solución, empleamos lo que se denomina el vecindario de la solución, que almacena todos los posibles cambios de los nodos seleccionados por los nodos disponibles. Lo que diferencia las dos búsquedas locales que se implementan en esta práctica es la manera en que exploramos estos vecindarios, pero debemos abordar otro tema antes de pasar a esto.

Una cosa a tener en cuenta en la búsqueda local es que tenemos que usar un cálculo de la función fitness factorizada. Esto es, cuando hacemos un cambio *(w,u)* por *(w,v)*, con *w* el índice del nodo a sustituir en el vector de nodos seleccionados, y *u,v* los nodos que son sustituidos pertenecientes al vector de nodos disponibles, calculamos la calidad de la nueva solución como:

$$\forall\, w \in Sel, \partial(w) = anterior(w) - d_{wu} + d_{wv}$$

Donde *anterior(w)* es la aportación de w a la dispersión, es decir, la suma de las distancias con el resto de los nodos seleccionados. Citando al seminario 2, “al añadir un elemento las distancias entre los que ya estaban en la solución se mantienen y basta con calcular la dispersión por el nuevo elemento al resto de elementos seleccionados”, y “al eliminar un elemento, las distancias entre elementos que se quedan en la solución se mantienen y basta con restar la distancia del elemento al resto de elementos en la solución”. Nos basaremos en esto para hacer nuestra heurística más adelante. Ahora sí, podemos explicar los dos algoritmos que implementamos en esta práctica.

Algoritmo de búsqueda local aleatoria

La primera manera de recorrer el vecindario es la más simple e intuitiva. Una vez que lo generamos, introduciendo todos los pares posibles *(w, u)*, se recorre aleatoriamente. El pseudocódigo de su función *optimize* es el siguiente:

// declaramos las variables
mdd_problem <- problem // cast
n <- tamaño del problema
m <- tamaño de la solución
matriz <- matriz de distancias entre nodos
evals <- 0
disponibles es un vector vacío

seleccionados es una solución aleatoria
fitness <- fitness(seleccionados)

```
para cada i desde 0 hasta n - 1 hacer:
        si i no está en seleccionados hacer:
                añadir i a disponibles

// creamos el vector de distancias con los otros nodos
distances es un vector vacío
para cada i en seleccionados hacer:
        suma = 0
        para cada j en seleccionados hacer:
                si i no es igual a j hacer:
                        suma += matriz[i, j]
        añadir suma a distances

vecindario es un vector vacío
para cada i desde 0 hasta m-1 hacer:
        para cada j en disponibles hacer:
                vecindario <- (i, j)
desordenar vecindario

// bucle principal
mientras siga habiendo vecinos y evals sea menor que maxevals, hacer:
        ind_seleccionado <- vecino.first
        valor_disponible <- vecino.second

        // calculamos las nuevas distancias con el cambio
        new_distances es un vector vacío
        para cada i desde 0 hasta m-1 hacer:
                si i no es ind_seleccionado hacer:
                        v <- seleccionados[i]
                        contrib <- distances[i] - matriz[v, seleccionados[ind_seleccionado]] +
                                                + matriz[v, valor_disponible]
                        añadir contrib a new_distances en la posicion i

        // calcular contribución del nuevo elemento
        para cada i desde 0 hasta m-1 hacer:
                si i no es ind_seleccionado hacer:
                        nueva_contrib += matriz[seleccionados[i], valor_disponible]
        añadir nueva_contrib a new_distances en la posicion [indice_seleccionado]

        // calculamos nuevo fitness
        new_fitness = max(new_distances) - min(new_distances)
        evals <- evals + 1
```

```
// si mejora
si new_fitness es mejor que fitness hacer:
        // hacer el cambio
        valor_eliminado <- seleccionados[ind_seleccinado]
        seleccionados[ind_seleccionado] <- valor_disponible
        para cada nodo disponible hacer:
                si valor_disponible está en la posición i entonces:
                        disponibles[i] <- valor_eliminado

        distances <- new_distances
        fitness <- new_fitness

        se vacía vecindario
        para cada i desde 0 hasta m-1 hacer:
                para cada j en disponibles hacer:
                        vecindario <- (i, j)
        desordenar vecindario


devolver {seleccionados, fitness, evals}
```

Una vez que encontramos una mejora, es decir, un par (i, j) que mejore la dispersión, la llevamos a cabo (actualizando los datos necesarios) y se reinicia el vecindario. Esta es la manera en que tenemos en cuenta el modelo “primero mejor”, ya que al reiniciar el vecindario (para la nueva solución) se pasa directamente a la siguiente iteración.

La solución aleatoria de la que partimos se genera con *CreateSolution*, ya explicado en las secciones anteriores.

Hemos señalado en negrita la manera en que recorremos el vecindario. En el caso de la búsqueda local aleatoria, como era esperado, lo desordenamos con *shuffle*, y se recorre uno a uno. Esta línea es lo único que cambia con respecto al código de la búsqueda local heurística.

Algoritmo de búsqueda local heurística

Como hemos mencionado antes, lo único que cambia de esta implementación de la búsqueda local es la manera en que se recorre el vecindario. Vamos a definir una función, *ordenaVecindario*, que una vez tengamos el vecindario generado, lo ordene según la heurística que voy a explicar a continuación. Esta es la línea que cambiamos respecto a la implementación de la búsqueda local aleatoria.

La dispersión se calcula como la diferencia entre el máximo de las distancias entre los nodos de la solución y el mínimo de las distancias entre los nodos de la solución. Si sustituimos primero el nodo que mayor distancia tiene con el resto, hay una probabilidad mayor de mejorar la dispersión que si sustituimos un nodo con otra distancia del resto. Por otra parte, si sustituimos primero el nodo que menor distancia tiene con el resto, también es más probable que mejore la dispersión, ya que estaríamos subiendo el mínimo. Hemos

probado estas dos heurísticas por separado, y el problema con ellas es que no funcionan bien si hay muchos valores altos y muy pocos bajos, o muy pocos valores altos y muchos bajos, respectivamente, porque aunque quitáramos el valor más alto, seguiría habiendo una diferencia grande con el segundo más alto y el mínimo (análogamente, con el máximo y el segundo más bajo para la otra heurística).

Una manera de solventar esto es implementar una mezcla de las dos heurísticas. Si vamos quitando, uno a uno, los nodos que provocan estas distancias máximas y mínimas del vecindario, la dispersión disminuirá independientemente de encontrarnos con alguna de las situaciones que habíamos comentado antes. El pseudocódigo de la ordenación del vecindario según esta heurística es el siguiente:

```
ordenar vecindario de mayor a menor distancia con el resto de los nodos
vecindario_aux es un vector de pares vacío
para cada i desde 0 hasta la mitad de la longitud de vecindario hacer:
        añadir i-ésimo elemento de vecindario a vecindario_aux
        añadir i-ésimo último elemento de vecindario a vecindario_aux
si vecindario tiene tamaño impar
        añadir elemento del medio del vecindario a vecindario_aux

vecindario <- vecindario_aux
```

Creamos el vecindario añadiendo, uno a uno, los elementos de los extremos. Así se recorre cambiando primero los máximos y los mínimos.

Estructura del código

El proyecto sigue la plantilla de PRADO. Se va a explicar directorio a directorio, indicando los cambios que hemos realizado.

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).

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:

- *greedy.h*
- *heuristicLS.h*
- *mddp.h*
- *randomLS.h*
- *randomsearch.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:

- *greedy.cpp*
- *heuristicLS.cpp*
- *mddp.cpp*
- *randomLS.cpp*
- *randomsearch.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, y hemos comentado antes, y el resto de archivos implementan el método *optimize* para cada algoritmo, según su nombre.

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 los 4 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}.

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 algoritmo implementado (*random, greedy, randomLS, heuristicLS*) 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 *resultados.txt.* Luego los he pasado al archivo excel de PRADO, y son los resultados que vamos a analizar.

Resultados

Random

Caso	*n*	*m*	Mejor coste
GKD-b_1_n25_m2	25	2	0
GKD-b_2_n25_m2	25	2	0
GKD-b_3_n25_m2	25	2	0
GKD-b_4_n25_m2	25	2	0
GKD-b_5_n25_m2	25	2	0
GKD-b_6_n25_m7	25	7	12,71796
GKD-b_7_n25_m7	25	7	14,09875
GKD-b_8_n25_m7	25	7	16,76119
GKD-b_9_n25_m7	25	7	17,06921
GKD-b_10_n25_m7	25	7	23,26523
GKD-b_11_n50_m5	50	5	1,9261
GKD-b_12_n50_m5	50	5	2,12104
GKD-b_13_n50_m5	50	5	2,36231
GKD-b_14_n50_m5	50	5	1,6632
GKD-b_15_n50_m5	50	5	2,85313
GKD-b_16_n50_m15	50	15	42,74578
GKD-b_17_n50_m15	50	15	48,10761
GKD-b_18_n50_m15	50	15	43,19609
GKD-b_19_n50_m15	50	15	46,41245
GKD-b_20_n50_m15	50	15	47,71511
GKD-b_21_n100_m10	100	10	13,83202
GKD-b_22_n100_m10	100	10	13,66434
GKD-b_23_n100_m10	100	10	15,34538
GKD-b_24_n100_m10	100	10	8,64064
GKD-b_25_n100_m10	100	10	17,20051
GKD-b_26_n100_m30	100	30	168,72959
GKD-b_27_n100_m30	100	30	127,09726
GKD-b_28_n100_m30	100	30	106,37919
GKD-b_29_n100_m30	100	30	137,45316
GKD-b_30_n100_m30	100	30	127,47974
GKD-b_31_n125_m12	125	12	11,74514
GKD-b_32_n125_m12	125	12	18,78893
GKD-b_33_n125_m12	125	12	18,5316
GKD-b_34_n125_m12	125	12	19,48833
GKD-b_35_n125_m12	125	12	18,11242
GKD-b_36_n125_m37	125	37	155,43477
GKD-b_37_n125_m37	125	37	198,89462
GKD-b_38_n125_m37	125	37	187,96703
GKD-b_39_n125_m37	125	37	168,5902
GKD-b_40_n125_m37	125	37	178,19374
GKD-b_41_n150_m15	150	15	23,34608
GKD-b_42_n150_m15	150	15	26,7895
GKD-b_43_n150_m15	150	15	26,75447
GKD-b_44_n150_m15	150	15	25,93559
GKD-b_45_n150_m15	150	15	27,77301
GKD-b_46_n150_m45	150	45	227,74931
GKD-b_47_n150_m45	150	45	228,6029
GKD-b_48_n150_m45	150	45	226,74534
GKD-b_49_n150_m45	150	45	226,40961
GKD-b_50_n150_m45	150	45	248,85662

Algoritmo Random			
Caso	Coste medio obtenido	Desv	Tiempo (s)
GKD-b_1_n25_m2	0,0000	0,00	2,91E-01
GKD-b_2_n25_m2	0,0000	0,00	2,84E-01
GKD-b_3_n25_m2	0,0000	0,00	2,89E-01
GKD-b_4_n25_m2	0,0000	0,00	2,93E-01
GKD-b_5_n25_m2	0,0000	0,00	3,07E-01
GKD-b_6_n25_m7	14	11,75	3,54E-01
GKD-b_7_n25_m7	19,9080	29,18	2,96E-01
GKD-b_8_n25_m7	19,8403	15,52	2,92E-01
GKD-b_9_n25_m7	21,0183	18,79	2,88E-01
GKD-b_10_n25_m7	29,3681	20,78	2,93E-01
GKD-b_11_n50_m5	4,6541	58,61	3,74E-01
GKD-b_12_n50_m5	4,6923	54,80	3,65E-01
GKD-b_13_n50_m5	6,0104	60,70	3,70E-01
GKD-b_14_n50_m5	6,0192	72,37	3,68E-01
GKD-b_15_n50_m5	5,1836	44,96	3,69E-01
GKD-b_16_n50_m15	137,6800	68,95	5,65E-01
GKD-b_17_n50_m15	140,9930	65,88	5,69E-01
GKD-b_18_n50_m15	126,6820	65,90	5,67E-01
GKD-b_19_n50_m15	159,0340	70,82	5,60E-01
GKD-b_20_n50_m15	141,6170	66,31	5,67E-01
GKD-b_21_n100_m10	66,6405	79,24	6,66E-01
GKD-b_22_n100_m10	63,3437	78,43	6,75E-01
GKD-b_23_n100_m10	47,9347	67,99	6,81E-01
GKD-b_24_n100_m10	50,3438	82,84	6,75E-01
GKD-b_25_n100_m10	58,2331	70,46	6,86E-01
GKD-b_26_n100_m30	551,1560	69,39	1,30E+00
GKD-b_27_n100_m30	541,8400	76,54	1,27E+00
GKD-b_28_n100_m30	536,7750	80,18	1,25E+00
GKD-b_29_n100_m30	533,0620	74,21	1,25E+00
GKD-b_30_n100_m30	533,0390	76,08	1,30E+00
GKD-b_31_n125_m12	102,6890	88,56	8,04E-01
GKD-b_32_n125_m12	82,6390	77,26	8,10E-01
GKD-b_33_n125_m12	83,1520	77,71	8,05E-01
GKD-b_34_n125_m12	91,8833	78,79	8,08E-01
GKD-b_35_n125_m12	86,7800	79,13	8,21E-01
GKD-b_36_n125_m37	677,4740	77,06	1,67E+00
GKD-b_37_n125_m37	831,2140	76,07	1,70E+00
GKD-b_38_n125_m37	757,6770	75,19	1,69E+00
GKD-b_39_n125_m37	702,5190	76,00	1,68E+00
GKD-b_40_n125_m37	717,9680	75,18	1,70E+00
GKD-b_41_n150_m15	154,0770	84,85	1,01E+00
GKD-b_42_n150_m15	150,0650	82,15	1,00E+00
GKD-b_43_n150_m15	154,3420	82,67	9,79E-01
GKD-b_44_n150_m15	134,5810	80,73	9,68E-01
GKD-b_45_n150_m15	130,2080	78,67	9,76E-01
GKD-b_46_n150_m45	993,3480	77,07	2,17E+00
GKD-b_47_n150_m45	946,5350	75,85	2,20E+00
GKD-b_48_n150_m45	871,4390	73,98	2,20E+00
GKD-b_49_n150_m45	972,4770	76,72	2,20E+00
GKD-b_50_n150_m45	951,1680	73,84	2,22E+00

Media Desv:	**60,96**
Media Tiempo:	**9,16E-01**

Greedy

Algoritmo Greedy			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
GKD-b_1_n25_m2	0,0000	0,00	2,42E-05
GKD-b_2_n25_m2	0,0000	0,00	2,57E-05
GKD-b_3_n25_m2	0,0000	0,00	2,64E-05
GKD-b_4_n25_m2	0,0000	0,00	2,37E-05
GKD-b_5_n25_m2	0,0000	0,00	2,32E-05
GKD-b_6_n25_m7	70	81,91	1,50E-04
GKD-b_7_n25_m7	46,2892	69,54	1,45E-04
GKD-b_8_n25_m7	45,2235	62,94	1,49E-04
GKD-b_9_n25_m7	51,0500	66,56	1,46E-04
GKD-b_10_n25_m7	75,2284	69,07	1,56E-04
GKD-b_11_n50_m5	25,2057	92,36	1,91E-04
GKD-b_12_n50_m5	32,1528	93,40	1,95E-04
GKD-b_13_n50_m5	32,1196	92,65	1,84E-04
GKD-b_14_n50_m5	30,6587	94,58	2,09E-04
GKD-b_15_n50_m5	27,2085	89,51	1,86E-04
GKD-b_16_n50_m15	200,6640	78,70	9,98E-04
GKD-b_17_n50_m15	174,7470	72,47	9,96E-04
GKD-b_18_n50_m15	173,0410	75,04	1,01E-03
GKD-b_19_n50_m15	179,7720	74,18	9,95E-04
GKD-b_20_n50_m15	151,8620	68,58	1,09E-03
GKD-b_21_n100_m10	80,8015	82,88	1,10E-03
GKD-b_22_n100_m10	72,0000	81,02	1,10E-03
GKD-b_23_n100_m10	80,0659	80,83	1,09E-03
GKD-b_24_n100_m10	53,9805	83,99	1,10E-03
GKD-b_25_n100_m10	71,6454	75,99	1,22E-03
GKD-b_26_n100_m30	425,2220	60,32	9,40E-03
GKD-b_27_n100_m30	433,7370	70,70	8,05E-03
GKD-b_28_n100_m30	461,9950	76,97	8,10E-03
GKD-b_29_n100_m30	365,7750	62,42	8,05E-03
GKD-b_30_n100_m30	343,4660	62,88	8,12E-03
GKD-b_31_n125_m12	61,3610	80,86	1,89E-03
GKD-b_32_n125_m12	78,4384	76,05	1,88E-03
GKD-b_33_n125_m12	88,5954	79,08	2,01E-03
GKD-b_34_n125_m12	79,5444	75,50	1,86E-03
GKD-b_35_n125_m12	85,7423	78,88	1,89E-03
GKD-b_36_n125_m37	570,6940	72,76	1,62E-02
GKD-b_37_n125_m37	504,4460	60,57	1,65E-02
GKD-b_38_n125_m37	619,2960	69,65	1,59E-02
GKD-b_39_n125_m37	525,9380	67,94	1,68E-02
GKD-b_40_n125_m37	394,1070	54,79	1,60E-02
GKD-b_41_n150_m15	104,9760	77,76	3,34E-03
GKD-b_42_n150_m15	122,3940	78,11	3,33E-03
GKD-b_43_n150_m15	117,2690	77,19	3,30E-03
GKD-b_44_n150_m15	117,7910	77,98	3,33E-03
GKD-b_45_n150_m15	106,8850	74,02	3,30E-03
GKD-b_46_n150_m45	568,5890	59,94	3,05E-02
GKD-b_47_n150_m45	551,4350	58,54	3,25E-02
GKD-b_48_n150_m45	659,0810	65,60	3,26E-02
GKD-b_49_n150_m45	536,6530	57,81	3,05E-02
GKD-b_50_n150_m45	496,6760	49,90	3,05E-02

Media Desv:	**66,25**
Media Tiempo:	**6,37E-03**

Búsqueda Local Aleatoria

Algoritmo Búsqueda Local Aleatoria			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
GKD-b_1_n25_m2	0,0000	0,00	3,62E-05
GKD-b_2_n25_m2	0,0000	0,00	3,44E-05
GKD-b_3_n25_m2	0,0000	0,00	3,41E-05
GKD-b_4_n25_m2	0,0000	0,00	3,38E-05
GKD-b_5_n25_m2	0,0000	0,00	3,33E-05
GKD-b_6_n25_m7	32	60,47	3,84E-04
GKD-b_7_n25_m7	29,0354	51,44	3,13E-04
GKD-b_8_n25_m7	52,3337	67,97	2,36E-04
GKD-b_9_n25_m7	37,6179	54,62	4,42E-04
GKD-b_10_n25_m7	41,1712	43,49	4,28E-04
GKD-b_11_n50_m5	17,3346	88,89	5,28E-04
GKD-b_12_n50_m5	16,2274	86,93	6,16E-04
GKD-b_13_n50_m5	14,5212	83,73	5,78E-04
GKD-b_14_n50_m5	14,7586	88,73	8,75E-04
GKD-b_15_n50_m5	17,3520	83,56	7,88E-04
GKD-b_16_n50_m15	133,5130	67,98	3,31E-03
GKD-b_17_n50_m15	119,9310	59,89	2,95E-03
GKD-b_18_n50_m15	82,0721	47,37	3,22E-03
GKD-b_19_n50_m15	98,0023	52,64	2,93E-03
GKD-b_20_n50_m15	83,5011	42,86	2,33E-03
GKD-b_21_n100_m10	41,7386	66,86	4,36E-03
GKD-b_22_n100_m10	41,9944	67,46	2,68E-03
GKD-b_23_n100_m10	52,9109	71,00	3,38E-03
GKD-b_24_n100_m10	40,6014	78,72	3,43E-03
GKD-b_25_n100_m10	39,5917	56,56	3,78E-03
GKD-b_26_n100_m30	400,9210	57,91	2,00E-02
GKD-b_27_n100_m30	309,1600	58,89	2,24E-02
GKD-b_28_n100_m30	374,3550	71,58	2,14E-02
GKD-b_29_n100_m30	288,7420	52,40	2,26E-02
GKD-b_30_n100_m30	382,0040	66,63	1,89E-02
GKD-b_31_n125_m12	51,6441	77,26	1,01E-02
GKD-b_32_n125_m12	48,9680	61,63	5,20E-03
GKD-b_33_n125_m12	43,5053	57,40	6,50E-03
GKD-b_34_n125_m12	45,5408	57,21	5,95E-03
GKD-b_35_n125_m12	49,1967	63,18	7,52E-03
GKD-b_36_n125_m37	417,4890	62,77	4,24E-02
GKD-b_37_n125_m37	421,6860	52,83	5,73E-02
GKD-b_38_n125_m37	427,4200	56,02	5,38E-02
GKD-b_39_n125_m37	420,8110	59,94	4,16E-02
GKD-b_40_n125_m37	383,8990	53,58	5,27E-02
GKD-b_41_n150_m15	73,0321	68,03	9,45E-03
GKD-b_42_n150_m15	67,1398	60,10	1,16E-02
GKD-b_43_n150_m15	58,3925	54,18	1,11E-02
GKD-b_44_n150_m15	77,0971	66,36	1,15E-02
GKD-b_45_n150_m15	66,2995	58,11	1,02E-02
GKD-b_46_n150_m45	467,3040	51,26	9,58E-02
GKD-b_47_n150_m45	440,1930	48,07	9,80E-02
GKD-b_48_n150_m45	478,8290	52,65	6,17E-02
GKD-b_49_n150_m45	443,2800	48,92	8,30E-02
GKD-b_50_n150_m45	522,9320	52,41	7,03E-02

Media Desv:	**55,81**
Media Tiempo:	**1,78E-02**

Búsqueda Local Heurística

Algoritmo Búsqueda Local Heuristica			
Caso	**Coste medio obtenido**	**Desv**	**Tiempo (s)**
GKD-b_1_n25_m2	0,0000	0,00	4,52E-05
GKD-b_2_n25_m2	0,0000	0,00	4,82E-05
GKD-b_3_n25_m2	0,0000	0,00	4,20E-05
GKD-b_4_n25_m2	0,0000	0,00	3,96E-05
GKD-b_5_n25_m2	0,0000	0,00	4,36E-05
GKD-b_6_n25_m7	29	55,81	5,43E-04
GKD-b_7_n25_m7	33,4005	57,79	3,84E-04
GKD-b_8_n25_m7	33,4445	49,88	4,22E-04
GKD-b_9_n25_m7	31,4396	45,71	4,47E-04
GKD-b_10_n25_m7	41,7093	44,22	4,40E-04
GKD-b_11_n50_m5	16,3649	88,23	7,15E-04
GKD-b_12_n50_m5	10,3249	79,46	7,27E-04
GKD-b_13_n50_m5	11,7661	79,92	7,28E-04
GKD-b_14_n50_m5	15,9237	89,56	7,13E-04
GKD-b_15_n50_m5	15,3281	81,39	6,61E-04
GKD-b_16_n50_m15	89,6831	52,34	4,39E-03
GKD-b_17_n50_m15	84,7674	43,25	4,55E-03
GKD-b_18_n50_m15	103,8940	58,42	3,96E-03
GKD-b_19_n50_m15	114,6380	59,51	2,96E-03
GKD-b_20_n50_m15	87,3115	45,35	3,31E-03
GKD-b_21_n100_m10	32,1683	57,00	5,38E-03
GKD-b_22_n100_m10	33,8624	59,65	6,08E-03
GKD-b_23_n100_m10	39,1924	60,85	4,88E-03
GKD-b_24_n100_m10	32,9483	73,78	5,63E-03
GKD-b_25_n100_m10	38,9017	55,78	5,90E-03
GKD-b_26_n100_m30	253,9720	33,56	3,07E-02
GKD-b_27_n100_m30	228,7160	44,43	3,16E-02
GKD-b_28_n100_m30	231,5670	54,06	3,04E-02
GKD-b_29_n100_m30	210,3830	34,67	3,31E-02
GKD-b_30_n100_m30	223,3160	42,92	3,15E-02
GKD-b_31_n125_m12	37,9776	69,07	1,04E-02
GKD-b_32_n125_m12	48,6829	61,41	1,07E-02
GKD-b_33_n125_m12	48,0150	61,40	9,01E-03
GKD-b_34_n125_m12	47,0815	58,61	9,61E-03
GKD-b_35_n125_m12	37,7973	52,08	8,85E-03
GKD-b_36_n125_m37	271,3280	42,71	5,86E-02
GKD-b_37_n125_m37	270,8300	26,56	7,23E-02
GKD-b_38_n125_m37	354,0020	46,90	4,69E-02
GKD-b_39_n125_m37	305,1390	44,75	6,18E-02
GKD-b_40_n125_m37	292,9280	39,17	6,52E-02
GKD-b_41_n150_m15	59,5386	60,79	1,60E-02
GKD-b_42_n150_m15	58,7462	54,40	1,95E-02
GKD-b_43_n150_m15	67,1553	60,16	1,73E-02
GKD-b_44_n150_m15	68,6951	62,25	1,61E-02
GKD-b_45_n150_m15	57,9392	52,07	1,52E-02
GKD-b_46_n150_m45	340,6060	33,13	1,14E-01
GKD-b_47_n150_m45	332,2230	31,19	9,50E-02
GKD-b_48_n150_m45	314,5110	27,91	8,74E-02
GKD-b_49_n150_m45	410,0690	44,79	9,11E-02
GKD-b_50_n150_m45	322,7120	22,89	1,04E-01

Media Desv:	**47,99**
Media Tiempo:	**2,28E-02**

Resultados globales por tamaño

Comparativa por tamaños				
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
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
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

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
Óptimo	66,4709	0,00	?

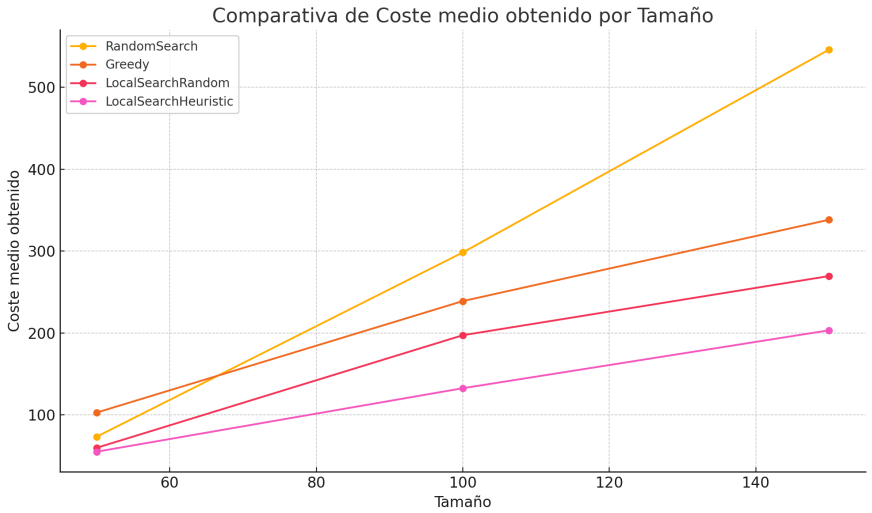
Análisis de resultados

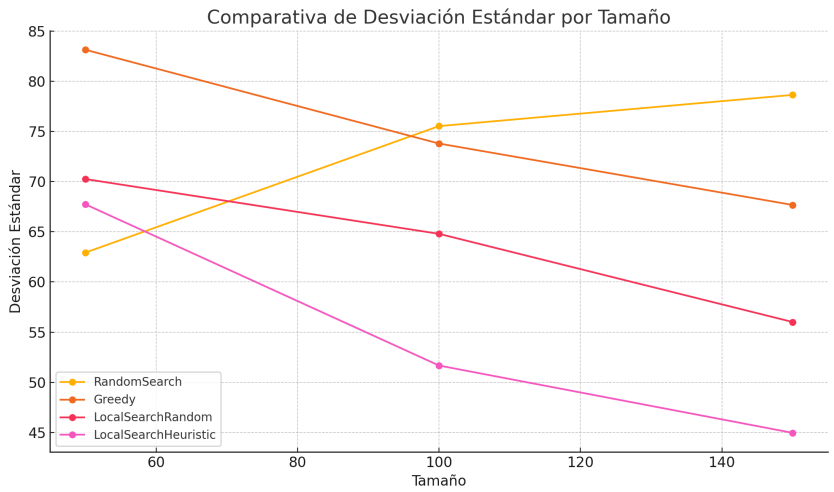
Una vez que tenemos los resultados, vamos a proceder a su análisis. Primero veremos los resultados globales para cada algoritmo, y después haremos un análisis más completo y específico.

Lo que primero salta a la vista son los resultados globales. La búsqueda local heurística es, con diferencia, el mejor algoritmo, con una desviación típica media de 47,99, mucho menor que cualquier otro, y tiene un coste medio bastante más bajo que el resto de algoritmos, algo que es lógico viendo el coste medio de los resultados óptimos. En términos generales, es el segundo que más tiempo tarda, pero no podríamos decir que es “lento”, ya que tarda solo centésimas de segundo en evaluar 5 veces los 50 archivos de texto.

La búsqueda local aleatoria es el segundo mejor algoritmo, con una desviación típica media de 55,81, en solo 0,0178 segundos. En tercer lugar tenemos el algoritmo aleatorio, y finalmente el algoritmo greedy. Lo único a remarcar de estos dos últimos casos es que son el algoritmo más lento y más rápido globalmente, respectivamente. Tendremos esto en cuenta más adelante.

Una vez analizados los resultados globales, vamos a analizar, otra vez algoritmo a algoritmo, y comparándolos, los resultados por tamaños.





Vemos directamente cambios importantes en cómo rinden los algoritmos ante problemas de distintos tamaños: ¡generar 100000 soluciones aleatorias consigue mejores resultados que cualquier otro algoritmo para $n = 50$! Esto tiene un coste, por supuesto, y es que tarde más de 100 veces más que el resto de algoritmos en calcular las soluciones. En cuanto aumentamos el tamaño del problema, vemos que realmente la solución aleatoria no es ni recomendable (por los resultados) ni realista (por el tiempo que emplea). Un detalle a tener en cuenta para este algoritmo es que la desviación típica es estable independientemente del tamaño de la solución, algo que no necesariamente pasa con el resto de algoritmos. Ya no vamos a hablar más del algoritmo aleatorio, ya que quitados los casos pequeños, es malo y muy lento.

Ahora con el greedy, vemos que produce resultados muy malos, especialmente con problemas de tamaño pequeño. Sin embargo, esto sí tiene más relevancia que el caso anterior ya que es algo concreto para nuestro problema (el algoritmo greedy es óptimo para problemas como la Mochila Fraccionaria, o Interval Scheduling). Entonces ¿por qué es tan malo para nuestro problema? Los algoritmos greedy, como hemos explicado en la sección

correspondiente, toman la mejor decisión posible para el estado actual, y en nuestro caso, eso genera que se quede estancado en máximos y mínimos locales. Lo que a primera impresión parecería ser una buena solución, porque en todo momento escoges el nodo que minimiza la dispersión, puede hacer que al final no te quede otra opción que tomar un nodo terrible, que haga la dispersión muy grande, como vemos en este problema. Los peores casos de este algoritmo son cuando hay que elegir muy pocos nodos entre no tantas opciones (n pequeño, m muy pequeño).
También hay que decir, a favor de este algoritmo, que para problemas grandes produce soluciones no tan malas (mejores que búsqueda local aleatoria en *GKD_b_50_n150_m45*) en muy, muy poco tiempo (la mitad de tiempo en *GKD_b_50_n150_m45*!). Para este problema, se podría considerar escoger el algoritmo greedy en muy pocos casos, a lo mejor cuando importara mucho mucho el tiempo de ejecución y se tuvieran problemas muy grandes.

Pasando a las búsquedas locales, vemos que casi siempre obtienen el mejor resultado. Curiosamente, incluso en tamaño 50, producen un coste medio menor que random, esto es, random funciona mejor en ciertos casos y mucho peor en otros casos. Por otra parte, son "rápidas" teniendo en cuenta la calidad de la solución. Concretando, la búsqueda local heurística produce mejores resultados que la aleatoria (lógicamente), aunque tarda algo más. Esto se debe a que se producen menos evaluaciones pero cada una tarda más, y por otra parte, la versión heurística tiene una complejidad de espacio mayor que la búsqueda local aleatoria, por tener que usar un vector auxiliar para ordenar el vecindario.

A la hora de la verdad, aunque tenga algo más de complejidad de tiempo y de espacio, la búsqueda local heurística va a merecer la pena en (casi) todas las situaciones por encima que la búsqueda local aleatoria, ya que las diferencias no son significativas. Además, los casos malos de las dos búsquedas locales son los mismos, tamaño de problema pequeño y tamaño de solución más pequeño todavía (similar a Greedy, y justamente donde el random funciona mejor). Simplemente los resultados son mucho mejores, y estos vienen por el hecho de habernos quitado primero los nodos que mayor y menor distancia tienen con el resto, en vez de evaluar nodos por evaluar (en la ejecución del *main* se ven las evaluaciones totales, y son mucho más altas en la búsqueda local aleatoria que en la búsqueda local heurística).

Como conclusión, si tuviéramos que elegir un único algoritmo para la resolución de este problema, el algoritmo aleatorio únicamente funciona bien con tamaños reducidos de problema y solución (aunque cuesta un tiempo considerable). En cualquier otro caso no lo consideraríamos una opción. Para el resto de tamaño de problemas, el greedy funcionaría bien si tuviéramos un tiempo muy limitado para conseguir los resultados, pero solo para valores muy, muy grandes de solución (donde las diferencias con el resto de algoritmos son menores). Entre las dos búsquedas locales, realmente no habría casi ningún caso donde escoger la versión aleatoria frente a la heurística, ya que las diferencias de tiempo de ejecución son minúsculas y los resultados sí son considerablemente mejores. La versión aleatoria tendrá su uso en la práctica 2 con la implementación de los algoritmos 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, además de posibles heurísticas a utilizar para la búsqueda local, en el siguiente paper: información adicional.