









PROBLEMA DE LA MÁXIMA DIVERSIDAD

3.a: Búsqueda Por Trayectorias

Javier Ramírez Pulido
javierramirezp@correo.ugr.es

3ºCSI GRUPO A-2
DNI: 20501292n HORARIO: VIERNES 17:30 – 19:30

INDICE

 <u>Breve descripción/formulación del problema</u>	3
 <u>Explicación procedimiento.....</u>	4
 <u>Explicación de problemas por trayectoria.....</u>	7
○ <u>Esquema de representación.....</u>	10
○ <u>Función objetivo.....</u>	11
 <u>Pseudocodigos.....</u>	12
 <u>Experimento y análisis de los resultados.....</u>	21
○ <u>Descripción de los casos del problema empleados.....</u>	21
○ <u>Resultados obtenidos.....</u>	22
○ <u>Análisis de resultados.....</u>	23
 <u>Referencias bibliográficas.....</u>	26

Descripción del problema

El problema de la Máxima Diversidad es un problema de optimización combinatoria que consiste en, partiendo de un conjunto de N elementos, formar agrupaciones con la mayor diversidad posible. Estas agrupaciones podrían tener diferentes dimensiones, pero en nuestro caso será de tamaño m (m elementos) y el conjunto original S deberá tener estrictamente más elementos que aquellos que se pretenden seleccionar ($N > m$).

Con diversidad nos referimos a la suma de las distancias entre los pares de elementos seleccionados. Esta distancia d_{ij} entre el elemento i y el elemento j se obtiene de la distancia euclidiana:

$$d_{ij} = \sqrt{\sum_{k=1}^t (a_{ik} - a_{jk})^2}.$$

Que se almacenará en una matriz $D = (d_{ij})$ de dimensión $n \times n$ que contiene las distancias entre ellos. Estas distancias cumplen que $d_{ij} = d_{ji}$ y que $d_{ii} = 0$. Las distancias dependen del contexto de la aplicación en que se utilicen. Este problema es NP-Completo, lo que quiere decir que tiene una alta complejidad computacional. De esta forma, el tiempo de resolución crece exponencialmente con el tamaño del problema.

La complejidad del problema viene dada por la combinatoria $\binom{n}{m}$ donde n es el número de elementos totales y m el número de elementos a escoger. Así, para un ejemplo de 10 elementos de los que buscamos seleccionar 3 maximizando la diversidad, tendríamos $\binom{10}{3} = 120$ soluciones posibles, lo cual es asequible para un algoritmo exacto, pero para casos en los que el número de soluciones es elevado deja de ser factible.

Otros tipos de medidas de la diversidad existentes son la similitud del coseno y la medida de similitud en grupos de solucionadores de problemas en los que se pueden obtener valores positivos y negativos, para los casos en los que se considere que este tipo de medida es muy restrictivo y no adecuada para algunas aplicaciones del problema.

Otros nombres que ha recibido el problema son *problema de k -partición* o *problema del particionamiento equitativo*.

Breve explicación del procedimiento considerado para desarrollar la práctica

El desarrollo de la práctica ha estado basado en los siguientes pasos tras recibir la clase de explicación:

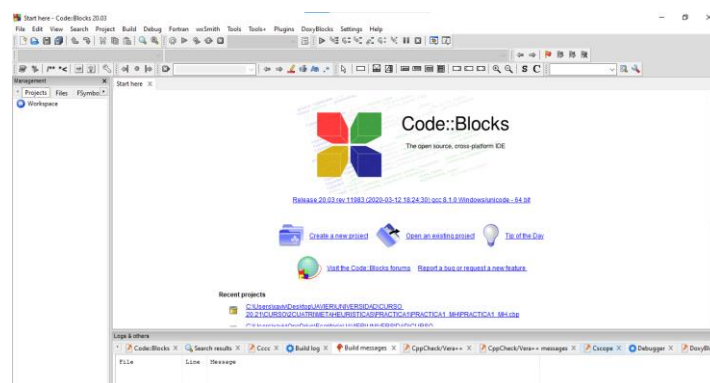
- Revisión del vídeo de explicación y aclaración del desarrollo de la práctica
- Lectura individual del pdf “Guion de la práctica 3.a: Problema de la Máxima Diversidad (MDP)”
- Lectura individual del pdf “Seminario 4: Problemas de optimización con técnicas basadas en trayectorias”
- Uso del código de BL realizado para la primera práctica.

En este caso, la implementación del código ha sido desde un inicio realizada por mí y, en algún momento inicial, consultada a algunas prácticas más de Algorítmica del curso anterior o al pseudocódigo de las diapositivas proporcionadas en la plataforma.

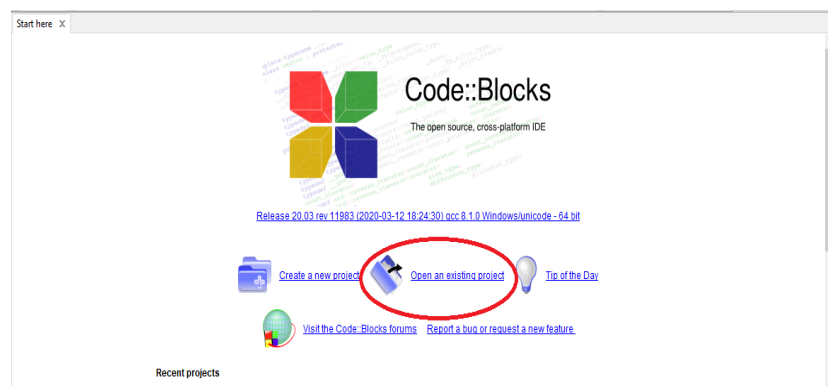
El lenguaje elegido ha sido C++ y el entorno de trabajo es el IDE Code::Blocks 20.03 debido a mi experiencia de uso en la asignatura de Inteligencia Artificial previamente.

La entrega del código se realizará en una carpeta llamada *software* que contendrá todos los archivos necesarios para su ejecución. La forma de realizar la correcta ejecución del programa será:

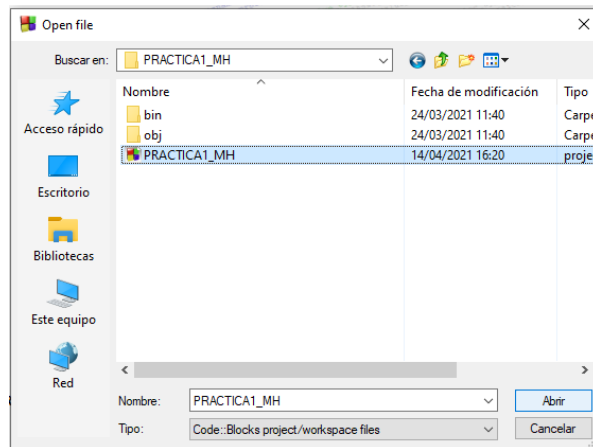
- Abrir CodeBlocks



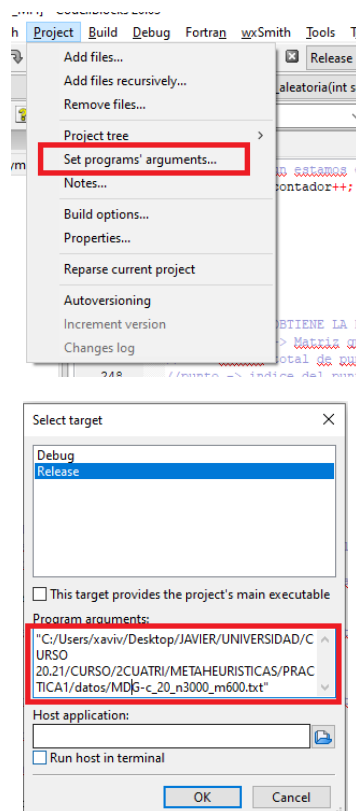
- Seleccionar *Open an existing Project*



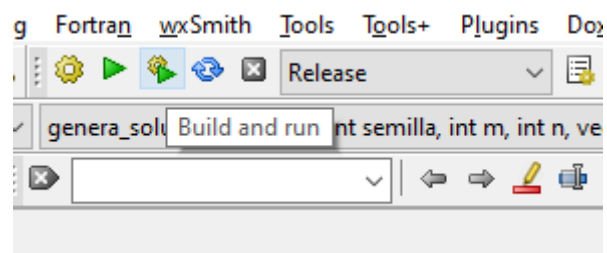
- Nos dirigimos al directorio en el que se encuentre el ejecutable .cbp y lo abrimos



- Una vez abierto, como argumento debemos meter la ruta del archivo del que queremos sacar los datos, que en mi caso sería
`"C:/Users/xaviv/Desktop/JAVIER/UNIVERSIDAD/CURSO
 20.21/CURSO/2CUATRI/METAHEURISTICAS/PRACTICA1/datos/archivo.txt"` [SEMILLA]



- Ya podemos construir y ejecutar el programa



Explicación de los problemas por trayectorias

Las metaheurísticas basadas en trayectorias tienen como metaheurística subordinada un algoritmo de búsqueda local que sigue una trayectoria en el espacio de búsqueda. Parten de una solución inicial e iterativamente tratan de reemplazarla por otra solución de mejor calidad. La búsqueda finaliza cuando se ha realizado un número máximo de iteraciones y se tiene una solución aceptable o cuando el proceso se atasca.

Estas heurísticas son más lentas, pero ofrecen mejor solución que las constructivas. Además, es un proceso de búsqueda realizado sobre un espacio de soluciones completas al problema.

Los algoritmos que se basan en trayectorias efectúan un estudio local del espacio de búsqueda, ya que analizan el entorno de la solución actual para decidir cómo continuar el recorrido de la búsqueda.

Para salir de óptimos locales pueden permitir movimientos de empeoramiento de la solución actual, modificar la estructura de entornos o volver a comenzar la búsqueda desde otra solución inicial.

Con búsquedas basadas en Trayectorias debería ser fácil generar una solución al problema, aunque en algunos problemas generar una solución válida puede ser de tipo NP-duro. Aun así, en estos casos es difícil aplicar un operador de vecino que garantice obtener una solución factible y puede ser mejor relajar las restricciones del problema pudiendo generar soluciones no válidas.

Desde cualquier solución de partida debería poder llegar hasta una solución óptima y cualquier solución del entorno debe estar relativamente cerca de esta.

La principal diferencia con las técnicas basadas en poblaciones es que las basadas en trayectorias hacen uso de una solución durante toda la búsqueda y es por lo que describe una trayectoria desde la solución de partida hasta la final, mientras que los genéticos hacen uso de un conjunto de soluciones que se optimizan simultáneamente durante la resolución.

La aplicación de estos algoritmos suele estar orientada a la resolución de problemas académicos con optimización combinatoria, además de aquellos mencionadas en prácticas anteriores para algoritmos de búsqueda.

Enfriamiento Simulado (ES): Este es un algoritmo de búsqueda por entornos con un criterio probabilístico de aceptación de soluciones basado en Termodinámica. Este algoritmo evita finalizar en óptimos locales permitiendo algunos movimientos hacia soluciones peores, pero si el avance está siendo hacia una buena solución estos empeoramientos deben controlarse. El enfriamiento simulado controla estos movimientos mediante una función de probabilidad que va disminuyendo con el avance de la búsqueda. La filosofía perseguida es diversificar al principio e intensificar al final (el modelo del embudo comentado en clase).

Este control se basa en el trabajo de Metrópolis en el campo de la termodinámica estadística en el que modela el proceso de enfriamiento simulando los cambios energéticos en un sistema de partículas conforme decrece la temperatura hasta alcanzar un estado estable. En este caso, si se mejora la solución se acepta directamente, pero si esta empeora, el cambio se aceptará con una probabilidad indicada por la expresión:

$$P_{\text{aceptacion}} = e^{(-\Delta \text{Costes} / (k \cdot t))}$$

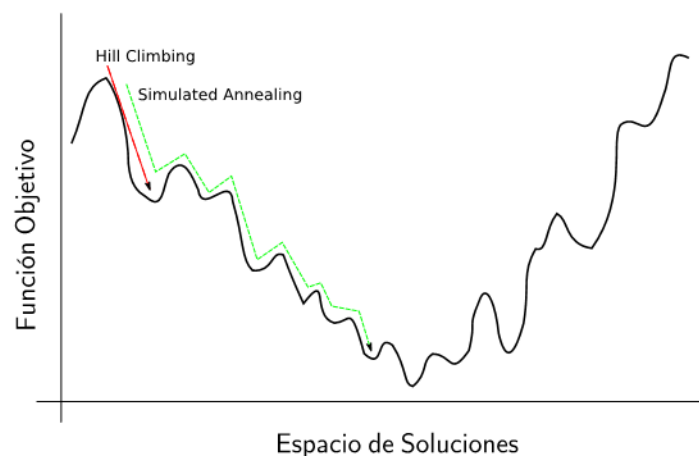
Aunque la k en nuestro caso, al ser una constante, se omite. Como se puede ver, el valor de la temperatura (que se va modificando en el tiempo) determina la probabilidad con la que se aceptan soluciones vecinas peores, pues cuanto mayor sea esta, más probabilidades hay.

Inicialmente la temperatura tiene un valor alto

$$T_0 = \frac{\mu \cdot C(S_0)}{-\ln(\phi)}$$

y en cada iteración se va enfriando mediante un mecanismo hasta alcanzar la temperatura final:

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



Búsqueda Multiarranque Básica (BMB): Es un algoritmo de búsqueda global que inicialmente genera una solución aleatoria S factible y a esta le aplica una BL para optimizarla. Este procedimiento se repite hasta cumplir un criterio de parada que, en nuestro caso, es el número de evaluaciones. Para ello, limitamos las evaluaciones de BL a 10000 y repetimos el procedimiento 10 veces. Finalmente devuelve la mejor solución encontrada en el proceso.

Las opciones con este algoritmo van desde la generación aleatoria de la solución inicial hasta el empleo de sofisticados métodos de construcción para partir de soluciones de calidad; avanzar en la búsqueda con algoritmos de búsqueda local básica o métodos más complejos y, para terminar, con un criterio de parada complejo según la evolución o guiado por el número de iteraciones.

Un problema de este algoritmo que mencionaremos en la obtención de los resultados es que al generar una solución aleatoria corremos el riesgo de que se generen muchos puntos cercanos y al aplicar BL se alcance siempre el mismo óptimo local.

Búsqueda Local Reiterada (ILS): El objetivo de ILS es enfocar la búsqueda en un pequeño espacio de soluciones que son soluciones óptimas locales.

Debemos partir de una solución inicial que puede ser generada aleatoriamente u obtenida de cualquier otro procedimiento que nos asegure un comienzo de calidad. Tras esto, se realiza la

búsqueda entre posibles soluciones que la mejoren (en nuestro caso que le aumenten la distancia entre los puntos) mediante la aplicación de algún movimiento en la vecindad.

Aunque existen muchas variaciones del algoritmo, como el criterio del mejor o primer vecino, prefijar un número de iteraciones para realizar una perturbación a la mejor solución obtenida o la forma de obtener la primera solución, voy a explicar en que consiste nuestra Búsqueda Local Reiterada implementada en la práctica.

Inicialmente se genera una solución aleatoria y se optimiza con BL. La mejor de estas es almacenada como la mejor solución obtenida hasta el momento y en un bucle de 9 iteraciones más partimos de la mejor solución, la mutamos aleatoriamente, le aplicamos BL y, de haber conseguido mejorar la mejor solución almacenada, esta se actualiza. En caso contrario, se vuelve a seleccionar la mejor almacenada, se vuelve a mutar aleatoriamente, se le aplica BL y volvemos a comprobar la calidad de esta modificada con respecto de la que partía.

Finalmente, el algoritmo devolvería la mejor solución que se encuentre en este procedimiento.

- **Mutación:** En este caso, la mutación consiste en seleccionar t ($t = 0.1 * m$) elementos aleatorios del vector de solución e intercambiarlos por t elementos aleatorios del vecindario, que es el motivo de la importancia que tiene que, además de almacenar la mejor solución hasta ahora, almacenemos también el vecindario de esta. Esto es debido a que en el procedimiento de modificación se parte de la mejor, el vecindario sufre una alteración en el proceso de mutación y BL y si la nueva solución no es mejor que la mejor que teníamos, en la siguiente iteración se parte de nuevo de la mejor y esta requiere de un vecindario reestablecido para empezar el procedimiento de mutación y BL de forma honesta.

Hibridación de ILS y ES: La idea es aplicar exactamente el mismo procedimiento anterior, replicando la generación de solución de partida y la mutación de t elementos, con la única diferencia de que en los momentos en los que se aplica una BL para optimizar la solución, se aplica el Enfriamiento Simulado implementado en esta práctica.

Como elemento común a los cuatro algoritmos tenemos una función que calcula el aporte de un punto en un conjunto de puntos. Esta función te permite a raíz de un punto y un vector, obtener cuál es la distancia que este aporta a la total del conjunto. Además, en base a su implementación, esta función acepta como entrada un punto que ignorar en el proceso de cálculo, lo cual nos permite calcular también el aporte que tendría un elemento que no se encuentre dentro si quisiéramos sustituirlo por otro. De esta forma, en el caso de querer saber qué aporte genera un punto i dentro del vector Solución, siendo i parte de este conjunto, no tendríamos mas que indicar que el punto a ignorar es i (porque su distancia con él mismo es 0 y no habría efecto negativo alguno, de hecho, nos quitamos los cálculos innecesarios) y calcularía la distancia de i a los $(m-1)$ elementos restantes. Por otra parte, si lo que queremos es saber cuánto aportaría un elemento j del vecindario si entrase al conjunto en sustitución de otro elemento, el punto a ignorar es ese elemento i que buscamos sustituir, obteniendo la distancia desde j a todos los puntos menos i , pues de mejorar el coste total, i dejaría de ser parte del conjunto. Una última alternativa que no se contempla usar en esta práctica es querer conocer el aporte que tendría un punto nuevo al ser insertado en Solución sin eliminar otro,

por lo que el punto a ignorar sería cualquiera que sepamos que no pertenece al conjunto de seleccionados, como un número negativo o mayor que n.

algoritmo calcula_aporte

variables locales

tipo double *aporte* <- 0

tipo entero *contador* <- 0

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *Solucion* (vector con los elementos de la solución)

tipo entero punto (es el punto del que calcular el aporte), punto_ignorar (punto que no tener en cuenta en el aporte, ya sea porque es igual que punto y tiene distancia 0 o porque sea un punto a sustituir)

inicio

1. *Recorremos todos los puntos punt de Solucion (contador < tamaño(Solucion))*

a. Si no es el punto a ignorar (punt != punto_ignorar)

i. Incrementamos 'aporte' con la distancia entre 'punto' y punt.

2. *Devolvemos el aporte de ese punto (Devolvemos aporte)*

Esquema de representación

El esquema de representación de los datos y la solución es un vector de enteros que contiene m elementos diferentes comprendidos entre 0 y n-1. Estos elementos representan los puntos que se seleccionan como solución y deben ser aquellos que mejor coste aporten. También se hace uso de otro vector de enteros que actúa como complementario y contiene los (n-m) elementos restantes que no se encuentran en el vector de solución. Este vector es útil en el desarrollo para tener almacenado cuál es el vecindario de la mejor solución, necesario para los casos en los que el vecindario es modificado aún sin saber si esta nueva solución que estamos generando es mejor que la anterior y ese vecindario va a ser definitivo. Por último, debido a que las distancias no son enteros, la distancia final de una solución se representa mediante una variable de tipo double que contiene el coste de este vector de m elementos seleccionados. Al igual que en la primera práctica, el contenido de la solución, del vecindario y del coste están en decimal y no en binario.

Función objetivo

La función objetivo en nuestro caso es la de maximizar el coste que puede proporcionar una solución cumpliendo las restricciones de tener m puntos seleccionados que no se repitan.

Esta función realiza un cálculo de la distancia entre todos los puntos seleccionados en la solución, sin repetir combinaciones, para ver la diversidad aportada por el conjunto de m elementos seleccionados. Esta función objetivo se utiliza para cada algoritmo un total de 100000 veces para que todos los algoritmos, independientemente de las generaciones que realicen, tengan las mismas oportunidades. Se llama 'calcula_coste' y es la siguiente:

algoritmo calcula_coste

variables locales

tipo double *distancia_total_solucion* <- 0

tipo entero *punto_seleccionado* <- 0, *punto_restante* <- 0

variables como parámetros

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *Solucion* (vector con los elementos de la solución)

inicio

1. *Mientras no hayamos recorrido todos los elementos de la solución (punto_seleccionado < tamaño(Solucion) - 1)*
 - a. *Nos situamos en el siguiente elemento del vector Solucion (punto_restante <- siguiente(punto_seleccionado))*
 - b. *Iteramos por todos los puntos desde punto_restante hasta el final (punto_restante < tamaño(Solucion))*
 - i. *Vamos sumando cada distancia entre punto_restante y punto_seleccionado y lo sumamos a distancia_total_solucion (distancia_total_solucion <- distancia_total_solucion + matriz_dist_{punto_seleccionado, punto_restante})*
2. *Devolvemos la distancia total de esa solución (Devolvemos distancia_total_solucion)*

Pseudocódigos

El primer pseudocódigo irá sobre la generación de candidatas. Estos son aquellos puntos que optan a formar parte de la solución al algún punto del desarrollo y en un principio siempre son entre 0 y n-1.

algoritmo inicializa_candidatas

variables como parámetros

tipo entero n (cantidad de puntos totales que tiene nuestro problema)

tipo vector de enteros &S (vector por referencia que contendrá todos los valores entre 0 y n-1 del que se sacarán los m elementos más adelante)

inicio

1. Para cada i en [0, n-1]
 - a. S.añadir(i)

(Se devuelve por referencia el valor)

Ahora vemos cómo funciona la generación de una solución aleatoria a raíz de las candidatas

algoritmo genera_solucion_aleatoria

variables como parámetros

tipo entero n (cantidad de puntos totales que tiene nuestro problema), m (cantidad de elementos que tiene que tener la solución al terminar de generarla)

tipo vector de enteros &S (vector por referencia que contendrá todos los valores entre 0 y n-1 del que se sacarán los m elementos y que al sacarlos se devolverá como vecindario), &Solucion (Vector inicialmente vacío en el que se devolverán como referencia los m elementos aleatorios seleccionados como solución)

variables locales

tipo booleano Añadir

tipo entero contador <- 0, lugar <- 0, aleat

inicio

2. Mientras no se cojan m elementos (contador < m)
 - a. Añadir <- verdad
 - b. Aleat <- aleatorio en [0, n-1]
 - c. Para cada d dentro de Solucion
 - i. Si algun d es aleat entonces Añadir <- mentira
 - d. Si Añadir es verdad
 - i. Para cada vec desde 0 hasta tamaño(S)
 1. Si S[vec] es aleat
 - a. Lugar <- vec y salimos del bucle
 - ii. Solucion.añadir(aleat)
 - iii. S.borrar(lugar)

(Se devuelven por referencia el vector solución v S que queda como vecindario de este)

Pseudocódigo de BL que incluye generación de vecino y exploración del entorno. La factorización se lleva a cabo en el cálculo del coste una única vez al tener el mejor conjunto y no en cada cambio. Por cada vecino que se explora o se intenta intercambiar solamente se calcula el aporte de un punto. Se ha arreglado una errata desde la práctica 1 que no supe resolver en la entrega de la segunda relacionada con el número de evaluaciones que se realizan.

algoritmo BL

variables locales

tipo double *total_distancia_punto_actual*, *aporte_mas_bajo*

tipo entero *índice_aporte_mas_bajo*, *simula_i*, *simula_j*

tipo booleano *seguir*

variables como parámetros

tipo entero *m* (número de elementos a seleccionar), *n* (número total de elementos), *contador* (número de veces que ha sido llamada la función recursivamente para no pasarnos del máximo)

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros *Solucion* (vector con los elementos de la solución), *S* (vector con los elementos no seleccionados)

inicio

1. *simula_i <- simula_j <- índice_aporte_mas_bajo <- 0, aporte_mas_bajo <- 1000000, seguir <- true*
(Inicializamos las variables)
2. *Bucle que recorra todos los elementos de la solución*
 - 2.1 *total_distancia_punto_actual <- sumatoria de la distancia del punto actual al resto de la solución*
 - 2.2 *Si total_distancia_punto_actual es menor que aporte_mas_bajo* (Si el punto actual aporta menos coste a la solución que el elemento que menos aportaba)
 - 2.1.2 *aporte_mas_bajo <- total_distancia_punto_actual* (Actualizamos la menor distancia que hemos visto hasta ahora)
 - 2.1.3 *índice_aporte_mas_bajo <- punto actual* (guardamos el índice del elemento)
3. fin del bucle

```

4. Bucle que recorre los elementos no seleccionados mientras 'seguir' sea verdadero (con esto buscamos de los
no seleccionados cual es el que mejora la solución si sustituye al elemento que menos aporta ya calculado)

4.1 Calculamos la distancia del elemento no seleccionado actual a todos los del conjunto de solución
excepto a aquel que queremos quitar

4.2 Si este elemento aporta más que el que menos aporta de la solución

4.2.1 seguir <- false (para que no busque más vecinos. No queremos el mejor, queremos el primero)

4.2.2 Solucion <- eliminar(indice_aporte_mas_bajo)

4.2.3 Solucion <- añadir(elemento actual)

4.2.4 S <- eliminar(elemento actual)

4.3 contador <- contador +1 (cuenta como evaluacion)

5. fin del bucle 6. Si seguir es falso (se ha encontrado ningún elemento que mejore la solución) y contador es
menor que 9999

6.1 llamamos recursivamente al algoritmo BL

7. Si seguir es verdadero o el contador ha llegado al máximo

7.1 Bucle que recorre el vector de Solución

7.1.1 Suma la distancia del punto actual con todos los posteriores para obtener el coste total

7.2 fin del bucle

Fin

```

La forma de calcular la temperatura y el modelo de enfriamiento no se realizan en una función aparte porque son fórmulas simples que se nos proporcionan, pero la explico como un procedimiento aparte para favorecer el pseudocódigo de ES. La fórmula y las recomendaciones:

$$T_0 = \frac{\mu \cdot C(S_0)}{-\ln(\phi)} \quad \text{se considerará } \phi = \mu = 0,3.$$

algoritmo Temperatura_inicial

variables como parámetros

tipo double coste (Coste de la primera solución generada aleatoriamente dentro del ES), &T (variable en la que se devolverá por referencia la temperatura inicial)

inicio

1. $T = 0.3 * \text{coste} / -\log(0.3)$

(Se devuelven por referencia el valor de la temperatura)

El modelo de enfriamiento sigue una fórmula que depende de alguna constante y de la temperatura anterior. Estas también son proporcionadas como material:

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

algoritmo Calculo_beta

variables como parámetros

tipo double temperatura_inicial (Temperatura inicial del ES), temperatura_final (Temperatura final del ES),
M (cantidad de iteraciones en las que se reducirá la temperatura)

inicio

1. *Devuelve (temperatura_inicial - temperatura_final) / (M * temperatura_inicial * temperatura_final)*

algoritmo Enfriamiento

variables como parámetros

tipo double beta (Constante necesaria para la fórmula del enfriamiento), &T (variable por referencia que contiene la temperatura actual y en la que se devuelve la temperatura actualizada)

inicio

1. *T = T / (1 + beta * T)*

(Se devuelven por referencia el valor de la temperatura)

algoritmo Enfriamiento_Simulado

variables como parámetros

tipo entero n (cantidad de puntos totales que tiene nuestro problema), m (cantidad de elementos que tiene que tener la solución al terminar de generarla), evaluaciones (nos va a indicar si tenemos que generar solución aleatoria o no porque este siendo llamada desde ILS)

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros $\&no_seleccionados$ (vector que contendrá el vecindario de la mejor solución), $\&Solucion$ (vector con m elementos seleccionados en el que se devolverá la mejor solución obtenida con este algoritmo)

variables locales

tipo entero $max_no_seleccionados \leftarrow 10 * m$, $aceptados \leftarrow 1$

tipo double $max_ exitos \leftarrow 0.1 * max_no_seleccionados$, $M \leftarrow evaluaciones / max_no_seleccionados$, To , Tf ,
Beta

inicio

3. Si evaluaciones es 100000
 - a. Genera_solucion_aleatoria($m, n, Solucion, no_seleccionados$)
4. Si no, se va a usar la Solucion pasada como parámetro para la optimizacion
5. Guardamos la mejor_solución $\leftarrow Solucion$
6. Guardamos el mejor_coste $\leftarrow calcula_coste(Solucion)$
7. Guardamos vecindario de la mejor solución $mejor_vecindario \leftarrow no_seleccionados$
8. Temperatura_inicial($To, mejor_coste$)
9. Si $10e-3 \geq To$
 - a. $Tf = 10e-5$
10. Si no entonces $Tf = 10e-3$
11. Beta = calculo_beta(To, Tf, M)
12. Mientras To sea mayor que Tf y mientras aceptados no sea 0
 - a. Aceptados $\leftarrow 0$
 - b. Contador $\leftarrow 0$
 - c. Mientras no se generen todas las soluciones vecinas posibles (contador $< max_no_seleccionados$) y mientras aceptados no llegue al máximo (aceptados $< max_ exitos$)
 - i. $Alea1 \leftarrow aleatorio[0, tamaño(Solucion)-1]$
 - ii. $Alea2 \leftarrow -aleatorio[0, tamaño(no_seleccionados)-1]$
 - iii. $Aporte1 \leftarrow calcula_aporte(alea1, Solucion, alea1, matriz_dist)$
 - iv. $Aporte2 \leftarrow calcula_aporte(alea2, Solucion, alea1, matriz_dist)$
 - v. Si $aporte2 > aporte1$
 1. Aceptados $\leftarrow aceptados+1$
 2. Intercambiamos el vecino
 3. Actualizamos el coste
 4. Si el coste $> mejor_coste$
 - a. Actualizamos mejor_solucion
 - b. Actualizamos mejor_coste
 - c. Actualizamos mejor_vecindario
 5. Enfriamiento($Beta, To$)
13. Solucion $\leftarrow mejor_solucion$
14. No_seleccionados $\leftarrow mejor_vecindario$
15. Devolvemos mejor_coste

algoritmo Busqueda_Multiarranque_Basica

variables como parámetros

tipo entero n (cantidad de puntos totales que tiene nuestro problema), m (cantidad de elementos que tiene que tener la solución al terminar de generarla)

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros &no_seleccionados (vector que contendrá el vecindario de la mejor solución), &Solucion (vector con m elementos seleccionados en el que se devolverá la mejor solución obtenida con este algoritmo)

variables locales

tipo vector de enteros mejor_solucion, vecindario_original

tipo double mejor_coste <- -1, coste

tipo entero contador <- 0

inicio

16. *inicializa_candidatas(vecindario_original)*
17. *No_seleccionados <- vecindario_original*
18. *Durante 10 iteraciones (contador < 10)*
 - a. *Vaciamos tanto Solucion como no_seleccionados*
 - b. *No_seleccionados <- vecindario_original*
 - c. *genera_solucion_aleatoria(m, n, Solucion, no_seleccionados)*
 - d. *aplicamos BL (Solucion, no_seleccionados)*
 - e. *Si el coste de BL es mejor que mejor_coste*
 - i. *Mejor_solucion = Solucion*
 - ii. *Mejor_coste = coste*
 - iii.
19. *Solucion = mejor_solucion*
20. *Devolvemos mejor_coste*

algoritmo mutacion

variables como parámetros

tipo double t (cantidad de intercambios a hacer entre Solucion y no_seleccionados)

tipo vector de enteros &no_seleccionados (vector que contendra el vecindario de Solucion), &Solucion
(vector con m elementos seleccionados en el que se hara el intercambio)

variables locales

tipo entero contador <- 0, aleat1, aleat2, tam_vecindario <- tamaño(no_seleccionados)

inicio

21. *Mientras no se hayan hecho t intercambios (contador < t)*

- a. *Aleat1 <- aleatorio[0, tamaño(solucion)-1]*
- b. *Aleat2 <- aleatorio[0, tam_vecindario-contador-1] (no generamos los que acabamos de sacar de Solucion)*
- c. *Intercambiamos_vecino(Solucion, no_seleccionados, aleat1, aleat2)*

algoritmo ILS

variables como parámetros

tipo entero n (cantidad de puntos totales que tiene nuestro problema), m (cantidad de elementos que tiene que tener la solución al terminar de generarla)

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros &no_seleccionados (vector que contendrá el vecindario de la mejor solución), &Solucion (vector con m elementos seleccionados en el que se devolverá la mejor solución obtenida con este algoritmo)

variables locales

tipo vector de enteros mejor_solucion, mejor_vecindario

tipo double mejor_coste <- -1, coste

tipo entero contador <- 0, t <- 0.1 * m

inicio

22. *genera_solucion_aleatoria(m, n , Solucion, no_seleccionados)*
23. *Aplicamos BL (Solucion, no_seleccionados, coste)*
24. *Mejor_coste = coste*
25. *Mejor_solucion = Solucion*
26. *Mejor_vecindario = no_seleccionados*
27. *Mientras no haga las otras 9 iteraciones (contador < 9)*
 - a. *Solucion <- mejor_solucion (partimos siempre de la mejor)*
 - b. *No_seleccionados <- mejor_vecindario*
 - c. *Mutacion(Solucion, t, no_seleccionados)*
 - d. *Aplicamos BL sobre la mutacion BL(Solucion, no_seleccionados, coste)*
 - e. *Si el coste después de BL es mejor que mejor_coste*
 - i. *Mejor_coste = coste*
 - ii. *Mejor_solucion = Solucion*
 - iii. *Mejor_vecindario = no_seleccionados*
28. *Solucion = mejor_solucion*
29. *Devolvemos mejor_coste*

algoritmo ILS_ES

variables como parámetros

tipo entero n (cantidad de puntos totales que tiene nuestro problema), m (cantidad de elementos que tiene que tener la solución al terminar de generarla)

tipo matriz de double *matriz_dist* (distancia entre todos los puntos)

tipo vector de enteros $\&no_seleccionados$ (vector que contendrá el vecindario de la mejor solución), $\&Solucion$ (vector con m elementos seleccionados en el que se devolverá la mejor solución obtenida con este algoritmo)

variables locales

tipo vector de enteros *mejor_solucion*, *mejor_vecindario*

tipo double *mejor_coste* $\leftarrow -1$, *coste*

tipo entero *contador* $\leftarrow 0$, $t \leftarrow 0.1 * m$

inicio

30. *genera_solucion_aleatoria*($m, n, Solucion, no_seleccionados$)
31. *Aplicamos ES* (*Solucion*, *no_seleccionados*, *coste*)
32. *Mejor_coste* = *coste*
33. *Mejor_solucion* = *Solucion*
34. *Mejor_vecindario* = *no_seleccionados*
35. *Mientras no haga las otras 9 iteraciones* (*contador* < 9)
 - a. *Solucion* \leftarrow *mejor_solucion* (*partimos siempre de la mejor*)
 - b. *No_seleccionados* \leftarrow *mejor_vecindario*
 - c. *Mutacion*(*Solucion*, t , *no_seleccionados*)
 - d. *Aplicamos BL sobre la mutacion ES*(*Solucion*, *no_seleccionados*, *coste*)
 - e. *Si el coste después de ES es mejor que mejor_coste*
 - i. *Mejor_coste* = *coste*
 - ii. *Mejor_solucion* = *Solucion*
 - iii. *Mejor_vecindario* = *no_seleccionados*
36. *Solucion* = *mejor_solucion*
37. Devolvemos *mejor_coste*

Experimento y análisis de los resultados

Descripción de los casos del problema empleados

Existen casos del problema que han sido estudiados con anterioridad para comprobar el funcionamiento de los algoritmos de resolución. A pesar de existir diferentes grupos de casos, nos centraremos en los utilizados en esta práctica, que son los casos MDG.

Los archivos de este tipo contienen en la primera línea dos valores, el primero de ellos que indica la cantidad de puntos totales con los que cuenta el problema y el segundo la cantidad de elementos que vamos a escoger para nuestra solución. A partir de ahí, cada una de las $n(n-1)/2$ líneas contiene 3 valores, los dos primeros enteros que marcan los índices de los puntos entre los que existe la distancia representada con el tercer valor, de tipo double, de la fila.

En nuestro caso, utilizamos 30 ficheros diferentes de los cuales 10 son del tipo MDG-a, los cuales tienen matrices de 500x500, $m = 50$ y distancias aleatorias en $[0,10]$; 10 son del tipo MDG-b con matrices de 2000x2000, $m = 200$ y distancias aleatorias en $[0,1000]$; y, por último, 10 son del tipo MDG-c con matrices de 3000x3000, de las cuales 2 tienen $m=300$, 3 tienen $m=400$, 3 con $m=500$ y 2 con $m=600$.

En el caso de la semilla, se utilizan los ficheros *random.cpp* y *random.h* aportados por los profesores y el valor con el que se inicializa es pasado como parámetro. En nuestro caso, las tablas se han rellenado usando como semilla el entero con valor 5.

Los parámetros considerados en tres de los cuatro algoritmos (BMB, ILS e ILS_ES) han sido los mismos:

- m (entero): número de 1's que debe tener 'genes' de cada cromosoma de la población (cantidad de elementos que se escogen por solución).
- n (entero): número de elementos en total que tiene nuestro vector 'genes' en cada cromosoma de la población.
- distancias (matriz de punteros de double): Matriz que contiene todas las distancias.
- Solucion (vector de enteros): Vector solución que contiene los m elementos seleccionados. Este se pasa como referencia para almacenar finalmente el mejor resultado obtenido en el procedimiento.
- No seleccionados (vector de enteros): Vector vecindario que contiene los (n-m) elementos que no se han seleccionado y que será útil para la búsqueda de soluciones vecinas

Además de estos parámetros, el algoritmo ES tiene uno extra:

- evaluaciones (entero): Es un entero que en nuestra práctica va a poder contener dos valores (100000 y 10000) por lo que podría verse como un booleano. Este indica el número de evaluaciones que queremos que realice ya que, cuando se utiliza el algoritmo como única técnica de obtención de resultados tiene que realizar las 100000 comprobaciones que realizan todos los algoritmos, pero al ser llamado desde la función ILS_ES el número de evaluaciones se reduce a 10000 porque este algoritmo ya realiza 10 iteraciones. De esta forma, si este parámetro vale 100000, entenderemos que tenemos que generar una solución aleatoria inicial de la que partir, sin embargo, si 'evaluaciones' vale 10000 no generamos nada, sino que utilizamos la solución pasada como parámetro para optimizarla.

Resultados obtenidos

Enfriamiento Simulado			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n500_m50	6464,2000	17,48	4,00
MDG-a_2_n500_m50	6508,5800	16,25	4,00
MDG-a_3_n500_m50	6425,3100	17,19	4,00
MDG-a_4_n500_m50	6559,0100	15,59	4,00
MDG-a_5_n500_m50	6509,5700	16,06	4,00
MDG-a_6_n500_m50	6543,1300	15,83	4,00
MDG-a_7_n500_m50	6496,4900	16,41	4,00
MDG-a_8_n500_m50	6554,3900	15,44	5,00
MDG-a_9_n500_m50	6416,3400	17,42	6,00
MDG-a_10_n500_m50	6484,7500	16,65	4,00
MDG-b_21_n2000_m200	10055500,0000	11,01	20,00
MDG-b_22_n2000_m200	10052600,0000	10,93	21,00
MDG-b_23_n2000_m200	10117900,0000	10,46	20,00
MDG-b_24_n2000_m200	10064000,0000	10,87	21,00
MDG-b_25_n2000_m200	10080600,0000	10,76	20,00
MDG-b_26_n2000_m200	10096500,0000	10,59	40,00
MDG-b_27_n2000_m200	10069200,0000	10,94	19,00
MDG-b_28_n2000_m200	10100800,0000	10,45	20,00
MDG-b_29_n2000_m200	10053200,0000	11,01	20,00
MDG-b_30_n2000_m200	10076400,0000	10,80	23,00
MDG-c_1_n3000_m300	22585600	9,24	60,00
MDG-c_2_n3000_m300	22643700	9,08	36,00
MDG-c_8_n3000_m400	40124100	7,63	49,00
MDG-c_9_n3000_m400	40144300	7,58	48,00
MDG-c_10_n3000_m400	40176700	7,59	82,00
MDG-c_13_n3000_m500	62715800	6,41	73,00
MDG-c_14_n3000_m500	62653500	6,46	67,00
MDG-c_15_n3000_m500	62723400	6,37	69,00
MDG-c_19_n3000_m600	90275700	5,60	86,00
MDG-c_20_n3000_m600	90238900	5,65	81,00

BMB			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n500_m50	7588,2800	3,13	19,00
MDG-a_2_n500_m50	7636,5000	1,74	18,00
MDG-a_3_n500_m50	7582,5500	2,28	18,00
MDG-a_4_n500_m50	7566,8800	2,62	19,00
MDG-a_5_n500_m50	7575,0900	2,32	15,00
MDG-a_6_n500_m50	7586,5300	2,41	18,00
MDG-a_7_n500_m50	7558,7600	2,74	18,00
MDG-a_8_n500_m50	7563,9100	2,41	17,00
MDG-a_9_n500_m50	7593,1800	2,28	22,00
MDG-a_10_n500_m50	7615,1200	2,12	18,00
MDG-b_21_n2000_m200	11123000,0000	1,57	1829,00
MDG-b_22_n2000_m200	11100400,0000	1,65	2063,00
MDG-b_23_n2000_m200	11141100,0000	1,41	2106,00
MDG-b_24_n2000_m200	11095200,0000	1,73	2167,00
MDG-b_25_n2000_m200	11106300,0000	1,68	2428,00
MDG-b_26_n2000_m200	11113600,0000	1,58	1823,00
MDG-b_27_n2000_m200	11125800,0000	1,59	2025,00
MDG-b_28_n2000_m200	11098000,0000	1,61	2225,00
MDG-b_29_n2000_m200	11104900,0000	1,70	1724,00
MDG-b_30_n2000_m200	11112700,0000	1,63	2750,00
MDG-c_1_n3000_m300	24565800	1,28	9731,00
MDG-c_2_n3000_m300	24550600	1,42	11483,00
MDG-c_8_n3000_m400	42927400	1,17	20323,00
MDG-c_9_n3000_m400	42937300	1,15	20900,00
MDG-c_10_n3000_m400	42971500	1,16	20355,00
MDG-c_13_n3000_m500	66392300	0,93	31965,00
MDG-c_14_n3000_m500	66389700	0,88	32840,00
MDG-c_15_n3000_m500	66400500	0,88	32987,00
MDG-c_19_n3000_m600	94867500	0,80	47061,00
MDG-c_20_n3000_m600	94841800	0,84	42577,00

ILS			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n500_m50	7690,7800	1,83	14,00
MDG-a_2_n500_m50	7595,7200	2,26	14,00
MDG-a_3_n500_m50	7656,6000	1,32	14,00
MDG-a_4_n500_m50	7645,8300	1,60	15,00
MDG-a_5_n500_m50	7671,9200	1,07	15,00
MDG-a_6_n500_m50	7721,9800	0,67	18,00
MDG-a_7_n500_m50	7696,7500	0,96	16,00
MDG-a_8_n500_m50	7673,8200	0,99	19,00
MDG-a_9_n500_m50	7695,3700	0,96	18,00
MDG-a_10_n500_m50	7679,4300	1,30	15,00
MDG-b_21_n2000_m200	11192600,0000	0,95	3310,00
MDG-b_22_n2000_m200	11200000,0000	0,77	4192,00
MDG-b_23_n2000_m200	11207900,0000	0,81	3118,00
MDG-b_24_n2000_m200	11161900,0000	1,14	2212,00
MDG-b_25_n2000_m200	11212600,0000	0,74	3185,00
MDG-b_26_n2000_m200	11183300,0000	0,97	3687,00
MDG-b_27_n2000_m200	11216300,0000	0,79	3758,00
MDG-b_28_n2000_m200	11221900,0000	0,51	3170,00
MDG-b_29_n2000_m200	11191000,0000	0,94	2528,00
MDG-b_30_n2000_m200	11181300,0000	1,02	3660,00
MDG-c_1_n3000_m300	24747200	0,55	13460,00
MDG-c_2_n3000_m300	24734300	0,69	13589,00
MDG-c_8_n3000_m400	43282200	0,36	25256,00
MDG-c_9_n3000_m400	43150100	0,66	23853,00
MDG-c_10_n3000_m400	43205900	0,62	23238,00
MDG-c_13_n3000_m500	66806800	0,31	34822,00
MDG-c_14_n3000_m500	66749500	0,34	32861,00
MDG-c_15_n3000_m500	66771900	0,33	33180,00
MDG-c_19_n3000_m600	95280000	0,37	50678,00
MDG-c_20_n3000_m600	95220500	0,44	48970,00

ILS ES			
Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n500_m50	6503,8500	16,98	10,00
MDG-a_2_n500_m50	6476,6000	16,66	4,00
MDG-a_3_n500_m50	6589,0900	15,08	5,00
MDG-a_4_n500_m50	6543,4800	15,79	6,00
MDG-a_5_n500_m50	6461,1300	16,69	4,00
MDG-a_6_n500_m50	6509,4200	16,26	4,00
MDG-a_7_n500_m50	6498,1500	16,39	5,00
MDG-a_8_n500_m50	6487,7500	16,30	4,00
MDG-a_9_n500_m50	6616,7400	14,84	6,00
MDG-a_10_n500_m50	6452,9200	17,06	4,00
MDG-b_21_n2000_m200	10095800,0000	10,66	22,00
MDG-b_22_n2000_m200	10119900,0000	10,34	21,00
MDG-b_23_n2000_m200	10129700,0000	10,36	30,00
MDG-b_24_n2000_m200	10059500,0000	10,91	42,00
MDG-b_25_n2000_m200	10135400,0000	10,27	22,00
MDG-b_26_n2000_m200	10107800,0000	10,49	24,00
MDG-b_27_n2000_m200	10078200,0000	10,86	21,00
MDG-b_28_n2000_m200	10100200,0000	10,46	22,00
MDG-b_29_n2000_m200	10152900,0000	10,13	40,00
MDG-b_30_n2000_m200	10083800,0000	10,73	40,00
MDG-c_1_n3000_m300	22624300	9,08	39,00
MDG-c_2_n3000_m300	22648300	9,06	35,00
MDG-c_8_n3000_m400	40252900	7,33	73,00
MDG-c_9_n3000_m400	40159500	7,55	46,00
MDG-c_10_n3000_m400	40192300	7,55	46,00
MDG-c_13_n3000_m500	62734500	6,39	99,00
MDG-c_14_n3000_m500	62765000	6,29	120,00
MDG-c_15_n3000_m500	62627000	6,52	112,00
MDG-c_19_n3000_m600	90238700	5,64	68,00
MDG-c_20_n3000_m600	90295400	5,59	131,00

Algoritmo	Desv	Tiempo (ms)
Greedy	9,22	717,59
BL	2,11	933,26
ES	11,46	30,60
BMB	1,69	9718,13
ILS	0,88	11096,17
ILS_ES	11,28	36,83

Análisis de resultados

Para el análisis partiremos de la comparación y el razonamiento de los costes obtenidos por los algoritmos implementados en esta práctica, seguido de un análisis similar acerca del tiempo de ejecución de estos. Por último, terminaremos haciendo un repaso con respecto de los algoritmos de la primera práctica.

Costes

En cuanto a costes, como podemos observar en la tabla, el algoritmo que mejores resultados ha obtenido es ILS mientras que, con un resultado prácticamente igual de incorrecto, tenemos el Enfriamiento Simulado y la hibridación de ILS con Enfriamiento Simulado. En cuanto a estos dos últimos resultados mencionados, el razonamiento que nos podía ayudar a saber previamente que estos resultados serían similares era que uno de ellos se apoya en los resultados del otro para avanzar en la búsqueda.

Visto más en profundidad, el problema principal de ES reside en que elementos que se han sacado de la solución, se meten inmediatamente en el vecindario, siendo candidatos a volver a entrar en la solución, por lo que se podría dar el caso de sacar y meter el mismo elemento varias veces y gastar evaluaciones vacías en soluciones ya comprobadas. Además, es el único algoritmo que probabilísticamente acepta soluciones peores a la actual, por lo que no siempre se asegura un avance hacia el óptimo, aunque te permita escapar de óptimos locales. Otro punto a tener en cuenta es que con seguridad no realiza todos los enfriamientos que se le permiten. Esto es debido a que existe otra condición de parada además de la de la temperatura por debajo de un valor concreto, que es la de no haber aceptado ninguna solución. Si para una temperatura concreta, esta es lo suficientemente baja como para poder haber encontrado una solución relativamente aceptable, la generación de soluciones vecinas (aleatoria y con posibilidad de repetir combinaciones) no encuentra una mejor y las peores no se aceptan porque la probabilidad depende de la temperatura, el algoritmo considera que no es rentable continuar la búsqueda. Todas estas limitaciones hacían previsible que el avance de este algoritmo no sería extremadamente rápido hacia el óptimo y el resultado estaría lejos de ser el mejor de los algoritmos implementados.

Siguiendo con lo visto en el párrafo anterior, si un algoritmo genera una solución aleatoria y mejora esta con una versión de ES que permite aún menos evaluaciones que la original, obviamente los resultados que va a obtener nunca son mucho mejores que los de ES. Esto es porque al aplicar ES sobre la aleatoria inicial sí que deberíamos obtener mejoras, pero al mutar y aplicar ES de nuevo, volvemos a tener las mismas limitaciones y el resultado sigue saliendo lejos de lo deseado.

A pesar de no ser el algoritmo con mejor desempeño, BMB obtiene resultados realmente aceptables, ya que aplica un algoritmo de BL sobre 10 soluciones aleatorias iniciales. Esto

permite que de forma aleatoria se pueda obtener una solución inicial cercana a la óptima, se le aplica BL y de ser la mejor, se almacena, por lo que el resto de iteraciones no nos hace perder este buen resultado.

Por último, podríamos ver que ILS es el mejor en diversidad debido a su filosofía de trabajar continuamente con la mejor solución de las que hemos visto. Partimos de una solución aleatoria, pero esta es optimizada con BL y nos quedamos con la mejor, a la que se le aplica de nuevo BL y así hasta 10 veces. Un problema que podríamos encontrar con este algoritmo es que, en las primeras iteraciones, al alcanzar una solución buena, aplicarle BL no modifique nada y solo nos haga perder tiempo de ejecución, pero para eso contamos con un operador de mutación. Esto permite que, aún partiendo de una solución que a priori es la mejor, podamos aleatoriamente acercarnos más al óptimo y que cuando apliquemos BL salgamos de óptimos locales. En el caso de que la mutación empeore el resultado lo suficiente como para que incluso después de BL sea peor que la que tenemos almacenada, simplemente ignoramos la solución, volvemos a partir de la mejor que tenemos, se le hace otra mutación diferente y probamos a mejorarla, con otro vecindario nuevo. Avanzar solo entre soluciones buenas y mejoradas hace que en 100000 evaluaciones obtengamos el mejor coste de los cuatro.

Tiempo

En cuanto a tiempo, los dos algoritmos que de media son más rápidos son aquellos que utilizan ES. Esto es por el hecho ya mencionado de que no realiza todas las iteraciones en la mayoría de los casos y porque la generación de soluciones vecinas es con el intercambio de un elemento aleatorio de vecinos con otro aleatorio de la solución. Esta es la diferencia con BL, en la que hay que recorrer m elementos buscando el de menor aporte y luego entre n elementos encontrar el primero que lo mejore, cuando en ES no hay iteraciones de búsqueda.

Por otra parte, que BMB haga uso de BL en 10 ocasiones le hace tener un coste bastante más alto que aquellos en los que los vecinos son explorados aleatoriamente, pero lo que realmente marca la diferencia es que ILS además de hacer uso en diez ocasiones diferentes del algoritmo de Búsqueda Local, tiene un operador de mutación que genera $0.2 * m$ números aleatorios para luego hacer un intercambio de estos entre la solución y el vecindario.

Como podemos ver se cumple que, en este caso concreto, el tiempo es inverso al coste obtenido, ya que aquellos más lentos obtienen unos resultados mejores y aquellos más rápidos peores. No siempre tiene que cumplirse esto, pero hemos analizado que este corte reducido se debe a que el avance hacia la mejora no es muy sofisticado ni profundo.

Comparación con Greedy y BL:

Vemos que los algoritmos nuevos que no hacen uso de BL obtienen peores resultados que Greedy. Esto podría estar justificado por que Greedy conforma la solución con el mejor punto al que opta en cada momento, mientras que ILS_ES y ES parten de una solución aleatoria y el avance no es muy prometedor. De haber partido de soluciones aleatorias cercanas al óptimo, seguramente Greedy sería el peor de los reflejados en la tabla, pero la aleatoriedad en la selección de m elementos difícilmente proporcionen resultados buenos de entrada. Por otra parte, los algoritmos que utilizan BL cuentan con que no solo lo aplican una vez, sino que repiten la optimización varias veces, ya sea a la mejor solución o a una nueva aleatoria, por lo que tienen muchas más oportunidades de obtener mejores resultados que una sola ejecución de BL. Algo obvio era que estos algoritmos son más lentos que la Búsqueda Local por

individual, ya que esta es utilizada varias veces. Como mínimo el tiempo debería ser \geq $\text{n}^\circ\text{veces_usado} * \text{tiempo_BL}$.

Bibliografía

- [Enfriamiento Simulado \(ugr.es\)](#)
- [Microsoft PowerPoint - Int-Metaheurísticas-CAEPIA-2009.ppt \[Modo de compatibilidad\] \(ugr.es\)](#)
- PDF:
<https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjmxxv3-he7wAhU0AWMBHXLvCIAQFjABegQIBRAD&url=https%3A%2F%2F Dialnet.unirioja.es%2Fdescarga%2Farticulo%2F5454193.pdf&usg=AOvVaw3az62aNiWfVz07oC5ghosK>
- PDF:
<https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiRoZ7Ih-7wAhW76uAKHb0CBSIQFjACegQIAhAD&url=https%3A%2F%2Fupcommons.upc.edu%2Fbitstream%2Fhandle%2F2117%2F191405%2Fint-metaheuriisticas-caepia-2009-5718.pdf&usg=AOvVaw2Dqte4eBZ8H4bmDPDDebvK>
- [Tema05-Multiarranque_I-MetodosBasicos_y_G-12-13.pdf \(ugr.es\)](#)
- [Algoritmo de búsqueda local iterada para la secuenciación en talleres de flujo y minimización de makespan. \(us.es\)](#)