

A dark blue vertical bar is positioned on the left side of the page. A blue arrow-shaped banner points to the right from this bar, containing the text 'Práctica1.a:'. Below the banner, several thin, curved lines in shades of blue and grey sweep upwards from the bottom left towards the center of the page.








Práctica1.a:

Técnicas de Búsqueda Local y Algoritmo Greedy para el Problema de la Máxima Diversidad

Javier Ramirez Pulido (20501292n)
3º CSI GRUPO 2

javierramirezp@correo.ugr.es
VIERNES 17:30 – 19:30

INDICE

 <u>Breve descripción/formulación del problema</u>	2
 <u>Breve descripción de la aplicación de los algoritmos empleados al problema. Consideraciones comunes.....</u>	3
 <u>Descripción en pseudocódigo de la estructura del método de búsqueda y operaciones relevantes de BL.....</u>	6
 <u>Descripción en pseudocódigo de la estructura del método de búsqueda y operaciones relevantes de Greedy.....</u>	8
 <u>Breve explicación del procedimiento considerado para desarrollar la práctica.....</u>	11
 <u>Experimento y análisis de los resultados.....</u>	13
○ <u>Descripción de los casos del problema empleados.....</u>	13
○ <u>Resultados obtenidos.....</u>	14
○ <u>Análisis de resultados.....</u>	15
 <u>Referencias bibliográficas.....</u>	16

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*.

Descripción de la aplicación de los algoritmos empleados al problema.

Los algoritmos empleados a este problema para esta práctica serán Greedy (método voraz) y la Búsqueda local.

Greedy es un algoritmo que construye una solución de un problema de optimización paso a paso como resultado de un conjunto de elecciones que deben ser factibles, localmente óptimas e irrevocables. Estas decisiones son tomadas en función de la información disponible en cada momento y no volverán a ser replanteadas de nuevo a lo largo de toda la resolución. No acostumbran a lograr la solución óptima, pero sí aproximaciones rápidas, además de ser fácilmente implementables.

Para este algoritmo necesitaremos de 6 elementos principales:

- Un conjunto de candidatos: Son los elementos seleccionables para la solución. Inicialmente son todos los que contiene el problema, pero a medida que estos se van seleccionando, el conjunto va menguando en cardinalidad.
- Una solución parcial: Son los elementos que han sido seleccionados. Inicialmente está vacío y se van añadiendo elementos conforme se elijan como los más óptimos posibles.
- Una función de selección: Es la encargada de determinar cuál de los candidatos del conjunto de seleccionables es el mejor para ser introducido en la solución parcial.
- Una función de factibilidad: Comprueba si es posible completar la solución parcial para alcanzar una solución del problema.
- Un criterio para definir lo que es una solución: Mira que la solución parcial resuelva el problema.
- Una función objetivo: Es el valor de la solución parcial que se ha obtenido.

Para aquellas situaciones en las que no encontramos un algoritmo greedy que aporte una solución óptima, podríamos obtener un mejor resultado considerando elementos descartados por el propio algoritmo. Esto no importa cuando se considera un factor clave el tiempo que se tarde en resolver un problema o cuando greedy es utilizado para encontrar una primera solución de la que partir para el uso de otra heurística.

La heurística es un procedimiento que nos aporta una solución aceptable a un problema mediante métodos sin justificación formal. Existen heurísticas para problemas concretos (NP) y otras de propósito general, como el enfriamiento simulado, la búsqueda tabú o los algoritmos bioinspirados.

Algunas aplicaciones claves sobre este algoritmo son problemas simples, el problema de la mochila, los árboles de expansión mínima, el cálculo de caminos mínimos y códigos de Huffman.

Dentro de los problemas simples, algunos ejemplos son el problema del cambio que consiste en formar una cantidad M de dinero empleando el mínimo número de ejemplares de monedas, la planificación de tareas para establecer el orden que ha de tener un procesador para minimizar el tiempo que los procesos se pasan esperando ser ejecutados o una variante de este que consiste en escoger un conjunto que contenga el máximo de actividades respetando la exclusión mutua.

Por otra parte, para el problema de la mochila tenemos el escenario de n objetos asociados a un peso y un valor y una mochila con una capacidad máxima. De esta forma, el objetivo sería determinar qué objetivos hay que colocar en la mochila de modo que el valor total que transporta sea el máximo sin sobrepasar el límite.

Con respecto a los problemas de árboles de expansión mínima nos encontramos un algoritmo genérico que construye el árbol a base de añadir aristas una a una a través de una función de selección. Por otra parte, tenemos el algoritmo de Kruskal, que parte de un subgrafo de G y añadiendo las aristas de G de valor mínimo construye un subgrafo que cumpla ser el MTS deseado. Por último, tenemos el algoritmo de Prim que hace crecer el conjunto de aristas de tal forma que siempre es un árbol. Esto lo consigue escogiendo la arista de menos peso tal que un extremo de ella es uno de los vértices seleccionados y el otro es uno que aún no lo ha sido.

Para los problemas de caminos mínimos tenemos el algoritmo de Dijkstra y el de reconstrucción de caminos. El primero de ellos parte con un conjunto de vértices vistos del que se conoce el camino mínimo de ellos al vértice inicial. Así, coge el vértice con la distancia mínima desde el punto inicial hasta otro de destino pasando por estos vértices vistos, de una forma similar al algoritmo de Prim con la expansión de mínimos. El segundo algoritmo mencionado calcula el peso de los caminos de peso mínimo, pero no mantiene la suficiente información como para saber qué vértices los forman. Puede ser modificado para almacenar información con la intención de recuperar los vértices que integran los caminos de peso mínimo.

Finalmente, greedy es habitualmente aplicado en la codificación de Huffman. Esta es una técnica para comprimir datos muy efectiva. El objetivo es descubrir cómo codificar los caracteres para reducir el espacio que ocupan utilizando un código binario.

La aplicación de este algoritmo a nuestro problema concreto pasa por la selección de un primer punto de partida alejado al máximo del resto y elegir posteriormente aquellos que aporten máxima diversidad. La solución obtenida de este problema es representada como un vector de m elementos que contiene el índice de cada elemento seleccionado. Finalmente, se recorre este calculando la sumatoria de la distancia de cada punto al resto del conjunto.

El otro algoritmo aplicado es la **Búsqueda Local**, el cual es la base de muchos métodos usados en problemas de optimización. Se puede simplificar como un proceso iterativo que empieza en una solución y la va mejorando con modificaciones locales. Parte de una solución inicial, la cual puede haber sido generada aleatoriamente como en nuestro caso y busca entre sus vecinos una mejor solución. Si la encuentra, reemplaza el elemento por aquel que es mejor y continúa con el proceso un número limitado de veces o hasta que no haya mejor solución.

De esta forma, una parte importante del algoritmo es el correcto diseño del vecindario, que está formado por todas las posibilidades de soluciones que se consideran en cada punto. Hay dos formas de buscar entre los vecinos: seleccionar el mejor vecino de todos o el que primero mejora la solución.

Una ventaja de este algoritmo es la velocidad a la que encuentra soluciones y una desventaja es que suele quedar atrapado en mínimos locales, dependiendo demasiado de la solución inicial. Este método es determinístico y sin memoria, por lo que, para una misma solución de

partida, siempre llegará a la misma solución final. Además, esto hace que el gasto de memoria sea mínimo). Este algoritmo contaría con una función que evalúe la calidad de la solución.

La función heurística aproxima la calidad de una solución, la optimiza (maximiza o minimiza) y puede tomar valores negativos o positivos. No existen restricciones para la función, pero sí para los elementos.

La representación de un problema de optimización para la aplicación de este algoritmo necesita una elección para la representación de datos. En nuestro caso, la estructura de datos en la que se devuelve la solución vuelve a ser un vector que contiene los m índices de los puntos seleccionados como solución final y una variable que contiene el coste total equivalente a la distancia entre todos ellos.

Además, el problema tendrá una función objetivo cuyo valor tratamos de minimizar o maximizar, una función que genere la solución inicial que puede ser aleatoriamente o a partir de otra metaheurística (como greedy) y una función que genere un estado sucesor al dado.

Algunas aplicaciones de este algoritmo serían aquellos de escalada, como escalada simple que busca cualquier operación que suponga una mejora respecto al padre o escalada por máxima pendiente que no escoge el primer movimiento, sino que elige el mejor de ellos desde la posición actual. Otros algoritmos inspirados en analogías físicas y biológicas serían *Simulated annealing* que usa el algoritmo de escalada estocástico inspirado en el enfriamiento de metales y los *algoritmos genéticos* que hacen uso del algoritmo de escalada paralelo inspirado en los mecanismos de selección natural.

Varias son las consideraciones comunes de estos algoritmos. Para empezar, ambos se basan en la obtención de una solución aceptable y rápida por delante de aquella que sea óptima. De esta forma, ninguna te asegura obtener el mejor resultado posible, pero sí hacerlo en un tiempo razonable. Dentro de estas restricciones, la búsqueda local por lo general obtiene mejores tiempos de ejecución ante los mismos problemas.

Por otra parte, la estructura de la representación de datos es la misma, en ambos casos se parte de un conjunto de soluciones en que, al terminar el algoritmo, queda almacenado el conjunto de puntos elegido como solución. Además, ambas hacen uso de los mismos elementos, que son una matriz con las distancias, un número de elementos totales, un número de elementos a escoger, un vector de soluciones y un vector de puntos candidatos.

Una forma de complementarlos pasaría por el uso del algoritmo de Greedy para obtener un conjunto rápido que sirva como solución y que este sea aquel de partida en el algoritmo de Búsqueda local.

Descripción en pseudocódigo de la estructura del método de búsqueda y operaciones relevantes de BL

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*(*índice_aporte_mas_bajo*)

4.2.3 *Solucion* <- *añadir*(elemento actual)

4.2.4 *S* <- *eliminar*(elemento actual)

5. fin del bucle

6. Si seguir es falso (se ha encontrado ningún elemento que mejore la solución) y contador es menor que 99999

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

El algoritmo calcula la distancia de cada punto de S con el resto de elementos y escoge como primer elemento de la solución aquel que devuelva el total más alto. Tras esto, busca por cada uno de los no seleccionados la distancia más corta desde este hacia alguno de los seleccionados. De todas estas distancias mínimas desde cada no seleccionado a algún seleccionado, elegimos el valor más alto y añadimos a la solución el punto con el que se obtenía esta distancia. Este proceso se repite recursivamente hasta haber seleccionado tantos elementos como indique m.

método Generación de soluciones aleatorias

variables locales

tipo entero contador, donde, x

tipo booleano noesta

variables como parámetros

tipo entero *semilla* (entero para inicializar la semilla del generador de números aleatorios), *m* (número de elementos a seleccionar) *n* (número total de elementos)

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

inicio

1. *Inicializamos la semilla*

2. *Bucle para generar tantos números aleatorios como indique m*

2.1 *Generamos en x un número aleatorio entre 0 y m-1*

2.2 Bucle que recorre todos los elementos de Solución

2.2.1 Si x corresponde a algún punto dentro de Solucion

2.2.1.1 Noesta <- false (indica que el número generado no ha sido seleccionado aún)

2.3 Fin de bucle

2.4 Si noesta

2.4.1 Bucle que recorre S buscando la posición de x en los no seleccionados

2.4.1.1 Si encontramos x almacenamos en *donde* su posición

2.4.2 Fin de bucle

2.4.3 Solucion <- añadir(x)

2.4.4 S <- eliminar(*donde*)

Fin

Descripción en pseudocódigo de la estructura del método de búsqueda y operaciones relevantes de Greedy

algoritmo Greedy

variables locales

tipo double *sumatoria_distancias*, *dist_mas_larga*, *menor_distancia*

tipo entero *punto_mas_distanciado*

variables como parámetros

tipo entero *m* (número de elementos a seleccionar), *n* (número total de elementos)

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. *sumatoria_distancias* <- *dist_mas_larga* <- *punto_mas_distanciado* <- 0 (Inicializamos las variables)
2. Si *Solucion* <- \emptyset (aún no hemos seleccionado ningún elemento)
 - 2.1 Desde 0 hasta *n* (Recorremos todos los puntos del conjunto)
 - 2.1.1 *Sumatoria_distancias* <- suma de cada distancia del punto actual a todos los demás
 - 2.1.2 Si *sumatoria_distancias* es mayor que *dist_mas_larga* (Si la sumatoria de este punto al resto es mayor que la mayor distancia que hemos visto hasta ahora)
 - 2.1.2.1 *punto_mas_distanciado* <- *punto actual* (guardamos el índice del elemento)
 - 2.1.2.2 *dist_mas_larga* <- *sumatoria_distancias* (guardamos la nueva mayor distancia observada)
 - 2.2 fin del bucle
 - 2.3 *Solucion* <- añadir(*punto_mas_distanciado*) (añadimos el más alejado al conjunto de soluciones)
 - 2.4 *S* <- eliminar(*punto_mas_distanciado*) (borramos el escogido de los candidatos)
3. Si *Solucion* $\neq \emptyset$ (Si ya hay algún elemento seleccionado en la solución)
 - 3.1 *dist_mas_larga* <- *punto_mas_distanciado* <- 0 (inicializamos las variables)
 - 3.2 Para *i* desde 0 hasta el último elemento de *S* (bucle que recorre cada elemento no seleccionado)
 - 3.2.1 *menor_distancia* <- 1000000, *sumatoria_distancias* <- 0 (inicializamos las variables para cada iteración)

3.2.2 Para j desde 0 hasta el último elemento de Solución (bucle que recorre todos los elementos escogidos en la solución)

3.2.2.1 Si $\text{matriz_dist}[i][j]$ es menor que menor_distancia (Si la distancia del punto i al j es la más corta hasta ahora)

3.2.2.1.1 $\text{menor_distancia} \leftarrow \text{matriz_dist}[i][j]$ (actualizamos el valor de la distancia más corta)

3.2.3 fin del bucle

3.2.4 Si menor_distancia es mayor que dist_mas_larga (si la distancia más corta de un punto no seleccionado a cualquiera de los puntos seleccionados es más larga que la más larga hasta ahora)

3.2.4.1 $\text{dist_mas_larga} \leftarrow \text{menor_distancia}$ (actualizamos el valor de la distancia más larga de entre las más cortas entre el punto no elegido j y alguno de los puntos elegidos en Solucion)

3.2.4.2 $\text{punto_mas_distanciado} \leftarrow i$ (guardamos el índice del elemento que es el más lejano hasta ahora)

3.3 fin del bucle

3.4 $\text{Solucion} \leftarrow \text{añadir}(\text{punto_mas_distanciado})$ (añadimos el más alejado al conjunto de soluciones de entre los más cercanos)

3.5 $S \leftarrow \text{eliminar}(\text{punto_mas_distanciado})$ (borramos el escogido de los candidatos)

4. Para i desde 0 hasta el penúltimo elemento de Solucion (bucle para añadir al coste total, el coste desde el último punto añadido al resto de ellos sin contarse a sí mismo)

4.1 Actualizamos el valor del coste total añadiendo la suma del punto nuevo a cada uno de los ya seleccionados.

5. Si la cantidad de elementos en Solucion es menor que m (Si aún no se han escogido todos los elementos deseados, se llama recursivamente a esta función de nuevo)

fin

El algoritmo parte de una solución de m elementos sin repetir generados aleatoriamente. Este se recorre en busca de aquel elemento cuya sumatoria de distancias hacia los demás de la solución es la menor, es decir, aquel que aporte menos distancia al coste total. Cuando este se obtiene, se mira de aquellos elementos en S (no seleccionados) uno por uno, el aporte que tendrían si entrasen en la posición del que menos aporta. Esto se obtiene calculando la distancia desde este punto no seleccionado a todos los de la solución excepto al que vamos a sustituir. Si esta distancia total es mayor que el aporte del elemento de la solución que menos aporta, hacemos el cambio y volvemos a empezar a comprobar desde el principio, buscando qué elemento es ahora el que menos aporta y quién de los vecinos daría un mayor coste

entrando en su lugar. Esta función se llama recursivamente si aún no hemos completado las 100000 comprobaciones o si se ha recorrido el vector de elementos no seleccionados y ninguno aporta mejor solución que el peor elemento de los seleccionados. Cuando alguna de estas dos condiciones hace que dejemos de llamar recursivamente, pasamos a calcular cual es el coste total de la solución obtenida, realizando el cálculo una sola vez y evitando tener que obtenerlo por cada elemento que se intercambia.

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:

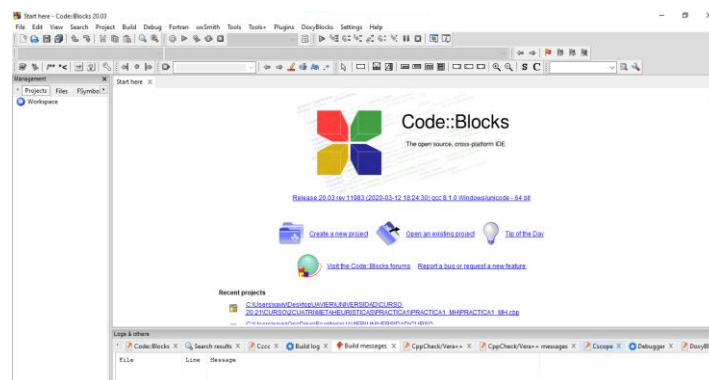
- Revisión del vídeo de explicación y aclaración del desarrollo de la práctica
- Lectura individual del pdf “Guion P1a LocalGreedy MDP MHs 2020-21”
- Lectura individual del pdf “Seminario 2.a. Trayectorias Simples”
- Extracción y uso de las instancias para el problema MDP y el generador de número aleatorios aportador en PRADO.

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.

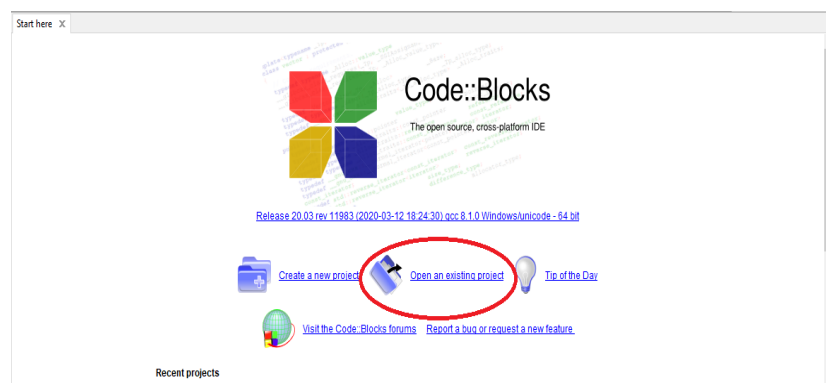
El lenguaje elegido ha sido C++ y el entorno de trabajo es el IDE Code::Blocks 20.03 debido a mi experiencia de uso en la asignatura de Inteligencia Artificial previamente.

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:

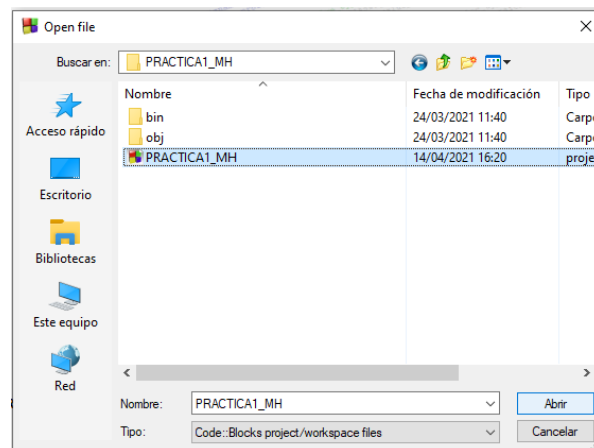
- Abrir CodeBlocks



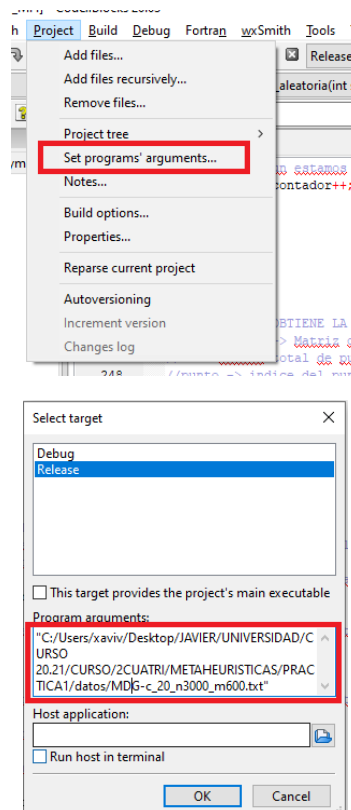
- Seleccionar *Open an existing Project*



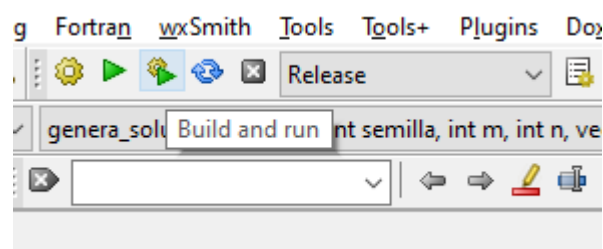
- Nos dirigimos al directorio en el que se encuentre el ejecutable .cbp y lo abrimos



- 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"`



- Ya podemos construir y ejecutar el programa



Experimentos y análisis de resultados

Descripción de los casos del problema empleados y de los valores de los parámetros considerados en las ejecuciones de cada algoritmo

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 2.

Los parámetros considerados en el algoritmo de **Greedy** han sido:

- m (entero): Cantidad de elementos a escoger para la solución
- matriz_dist (matriz de punteros de double): Matriz que contiene todas las distancias.
- n (entero): Número de elementos totales del problema (la matriz será $n \times n$)
- Solucion (Vector de enteros): Vector por referencia en el que se guardaran los puntos seleccionados como solución del problema
- S (Vector de enteros): Vector que contiene los puntos no seleccionados y del que se irán borrando conforme estos se cojan (también por referencia para los borrados)
- distancia_total (double): Variable por referencia que contiene la distancia total entre los puntos seleccionados una vez termina la función.

Los parámetros considerados en el algoritmo de **BL** han sido:

- m (entero): Cantidad de elementos a escoger para la solución
- matriz_dist (matriz de punteros de double): Matriz que contiene todas las distancias.
- n (entero): Número de elementos totales del problema (la matriz será $n \times n$)
- Solucion (Vector de enteros): Vector por referencia en el que se guardaran los puntos seleccionados como solución del problema
- S (Vector de enteros): Vector que contiene los puntos no seleccionados y del que se irán borrando conforme estos se cojan (también por referencia para los borrados)
- distancia_total (double): Variable por referencia que contiene la distancia total entre los puntos seleccionados una vez termina la función.
- Contador (entero): Variable que contendrá el número de veces que se ha llamado a la función de forma recursiva para no hacerlo más veces de las establecidas como máximas.

Resultados obtenidos según el formato especificado

Algoritmo GREEDY				Algoritmo BL			
Caso	Coste obtenido	Desv	Tiempo (ms)	Caso	Coste obtenido	Desv	Tiempo (ms)
MDG-a_1_n500_m50	6865,9400	12,36	2,99	MDG-a_1_n500_m50	7544,7200	3,69	1,99
MDG-a_2_n500_m50	6754,0200	13,09	2,99	MDG-a_2_n500_m50	7480,5000	3,75	2,00
MDG-a_3_n500_m50	6814,3500	12,18	2,99	MDG-a_3_n500_m50	7506,7000	3,26	2,00
MDG-a_4_n500_m50	6841,5900	11,95	2,99	MDG-a_4_n500_m50	7569,5300	2,58	1,99
MDG-a_5_n500_m50	6740,3400	13,09	2,99	MDG-a_5_n500_m50	7552,3200	2,62	2,99
MDG-a_6_n500_m50	7013,9400	9,77	4,00	MDG-a_6_n500_m50	7583,3600	2,45	2,00
MDG-a_7_n500_m50	6637,4600	14,59	3,99	MDG-a_7_n500_m50	7496,5300	3,54	2,99
MDG-a_8_n500_m50	6946,2800	10,38	2,99	MDG-a_8_n500_m50	7514,6600	3,05	3,00
MDG-a_9_n500_m50	6898,0100	11,22	2,99	MDG-a_9_n500_m50	7454,0900	4,07	1,95
MDG-a_10_n500_m50	6773,4600	12,94	2,99	MDG-a_10_n500_m50	7497,2900	3,64	2,07
MDG-b_21_n2000_m200	10314600,0000	8,72	270,06	MDG-b_21_n2000_m200	11105300,0000	1,72	263,65
MDG-b_22_n2000_m200	10283300,0000	8,89	261,38	MDG-b_22_n2000_m200	11064000,0000	1,97	222,93
MDG-b_23_n2000_m200	10224200,0000	9,52	289,48	MDG-b_23_n2000_m200	11071400,0000	2,02	213,49
MDG-b_24_n2000_m200	10263600,0000	9,10	267,29	MDG-b_24_n2000_m200	11090500,0000	1,77	247,30
MDG-b_25_n2000_m200	10250100,0000	9,26	268,77	MDG-b_25_n2000_m200	11063800,0000	2,06	243,31
MDG-b_26_n2000_m200	10196200,0000	9,71	266,35	MDG-b_26_n2000_m200	11101400,0000	1,69	263,05
MDG-b_27_n2000_m200	10358200,0000	8,38	269,34	MDG-b_27_n2000_m200	11091000,0000	1,90	204,96
MDG-b_28_n2000_m200	10277400,0000	8,89	266,30	MDG-b_28_n2000_m200	11052300,0000	2,02	251,91
MDG-b_29_n2000_m200	10291300,0000	8,90	200,00	MDG-b_29_n2000_m200	11067800,0000	2,03	266,30
MDG-b_30_n2000_m200	10263900,0000	9,14	271,27	MDG-b_30_n2000_m200	11111200,0000	1,64	251,34
MDG-c_1_n3000_m300	22943100	7,80	939,58	MDG-c_1_n3000_m300	24560300	1,30	1050,42
MDG-c_2_n3000_m300	22982400	7,72	931,21	MDG-c_2_n3000_m300	24531500	1,50	1141,37
MDG-c_8_n3000_m400	40434500	6,91	1641,37	MDG-c_8_n3000_m400	42884200	1,27	2142,97
MDG-c_9_n3000_m400	40488300	6,79	1530,66	MDG-c_9_n3000_m400	42873600	1,30	2092,25
MDG-c_10_n3000_m400	40455400	6,95	1526,79	MDG-c_10_n3000_m400	42806200	1,54	1800,91
MDG-c_13_n3000_m500	63170800	5,73	2176,80	MDG-c_13_n3000_m500	66346500	1,00	3164,46
MDG-c_14_n3000_m500	62817700	6,21	2221,06	MDG-c_14_n3000_m500	66227600	1,12	3156,72
MDG-c_15_n3000_m500	63066400	5,86	2176,29	MDG-c_15_n3000_m500	66332200	0,99	3090,78
MDG-c_19_n3000_m600	90566200	5,30	2876,12	MDG-c_19_n3000_m600	94778900	0,89	3931,47
MDG-c_20_n3000_m600	90602300	5,27	2845,73	MDG-c_20_n3000_m600	94772200	0,91	3975,36

*

Media Desv:	9,22
Media Tiempo:	717,59

Media Desv:	2,11
Media Tiempo:	933,26

Algoritmo	Desv	Tiempo
Greedy	9,22	717,59 ms
BL	2,11	933,26 ms

*El dato del tiempo y de la media del tiempo está en ms

Análisis de resultados

La desviación típica o estándar es una medida que representa que tan cerca están los datos de una muestra de su media. Cuanto más bajo es su valor, más datos se encuentran cercanos al valor esperado y mejor será el obtenido.

Sabiendo esto, de la tabla comparativa obtenemos la información de que, de media, el algoritmo de Búsqueda Local obtiene desviaciones mucho más bajas que el algoritmo de Greedy, lo que significa los resultados con este algoritmo son más cercanos al mejor coste proporcionado como referencia.

Analizando con más detalle la evolución de ambos algoritmos conforme aumentamos la cantidad de datos de partida o de selección, podemos ver que la búsqueda local es el algoritmo que mejor se adapta al cambio. La desviación media de Greedy de los 10 primeros archivos de datos es 12.16, mientras que los últimos 10 conjuntos de datos resultan en una media de 6.45, un 53% mejor. A su vez, Búsqueda Local tiene una media en los primeros conjuntos de datos de 3.27 y en los últimos de 1.18, con lo que mejoran en un 36%.

En cuanto a tiempo, el algoritmo Greedy obtiene una media por debajo de la búsqueda local. No solamente eso, sino que el pico más alto de tiempo es obtenido por Búsqueda Local con el conjunto "MDG-c_20_n3000_m600" con casi 4 segundos, mientras que por parte de Greedy, para el mismo archivo, el tiempo de cálculo no supera los 3 segundos, siendo el tiempo más lento de este algoritmo de 2876.12 ms.

Como caso extraordinario a comentar, cabe destacar que el único conjunto de datos de los 10 primeros con los que se obtiene una desviación por debajo de 10 con el algoritmo de Greedy es "MGD-a_6_n500_m50", casualmente aquel con el que se obtiene la desviación más baja de las 10 primeras con búsqueda local.

Una explicación teórica que puede estar detrás de estos resultados puede ser el procedimiento que sigue cada algoritmo en sí. En el caso de Greedy, este coge el mejor en cada momento sin tener en cuenta lo que haya ocurrido previamente ni lo que vaya a ocurrir en adelante, lo cual puede provocar que soluciones mejores no sean consideradas. Por otra parte, Búsqueda Local parte de una solución aleatoria que por azar puede andar cerca de la solución óptima, y a partir de esta, los cambios están destinados a aquellos elementos que renta más cambiar, lo que provoca que los resultados de este algoritmo tiendan a ser mejores.

Por motivos de eficiencia y computabilidad, Búsqueda Local cambia el elemento que menos aporta por el primero que encuentra que lo mejore y no por el que más lo mejore. Esto resultaría en una desviación aún menor, pero un tiempo demasiado elevado.

La explicación del tiempo que necesita el segundo algoritmo en comparación con Greedy es por la cantidad de bucles que existe dentro de la implementación. Mientras que Greedy itera en un conjunto que va menguando (el de los puntos candidatos), Búsqueda Local recorre todo el conjunto de la solución buscando el elemento con menor aporte y luego todo el conjunto de candidatos buscando uno que mejore el anterior, lo cual requiere calcular cuánto aportaría este y cuánto aportaría el que se encuentra en los seleccionados. De esta forma, la exploración de todo el vecindario ofrece $m(n-m)$ posibles cambios, lo cual puede requerir un coste elevado en eficiencia.

Referencias bibliográficas

[Tema 6: Búsqueda local y algoritmos genéticos \(us.es\)](#)

[2-BH3-Busqueda local.pdf \(upc.edu\)](#)

[4.2 Búsqueda Local \(Local Search\) \(inaoep.mx\)](#)

[Microsoft Word - GREEDY \(upc.edu\)](#)

[Microsoft PowerPoint - 4 Greedy.pptx \(ugr.es\)](#)

[TÉCNICAS HEURÍSTICAS Y METAHEURÍSTICAS PARA EL PROBLEMA DE LA MÁXIMA DIVERSIDAD \(MAXIMUM DIVERSITY PROBLEM \(MDP\)\)](#)

[maeb2012.dvi \(albacete.org\)](#)