PRÁCTICA 4 – Backtracking y Branch-and-Bound

Un problema de carga marítima

Objetivos

- Construir soluciones a un problema utilizando los **métodos algorítmicos *backtracking*** (vuelta atrás) y ***branch-and-bound*** (ramificación y poda). Analizar y comparar los algoritmos implementados desde diferentes perspectivas.
- Estudiar el **análisis de la eficiencia** de las soluciones aportadas, y una comparativa tanto desde el punto de vista teórico como práctico.

Requerimientos

Para superar esta práctica se debe realizar lo siguiente:

- Recordar (de EDA I) las **estructuras de datos** más apropiadas para la resolución de problemas.
- Conocer cómo implementar los **esquemas algorítmicos *backtracking*** y ***branch-and-bound*** para responder a una necesidad concreta (en este caso, resolver el problema del viajante de comercio y de ciclos hamiltonianos).
- **Evaluar los algoritmos implementados** siguiendo los métodos ***backtracking*** y ***branch-and-bound***, relacionando ambos esquemas algorítmicos.

Enunciado del problema

Uno de los puertos más importantes de EDALand es el de Almería debido a que últimamente está apostando por el transporte marítimo de verduras y hortalizas. Debido a ese incremento en el tráfico por mar y que el Corredor-Mediterráneo es más una promesa política que una realidad, **EDAPort-Almeria** está recibiendo diariamente muchos *buques portacontenedores* para poder exportar la riqueza más importante de esta provincia. Con el objetivo de que estos buques salgan del puerto con la mayor carga posible, respetando su peso o Capacidad Máxima de Carga (CMC), EDAPort-Almeria ha encargado a **theBestSoft (EDASoft)** que realice una aplicación para que **optimice el llenado de los buques en función de su CMC y, del peso y número de los contenedores que se pueden incluir en su carga.**

Para ello tenemos el siguiente planteamiento. Se va a cargar un buque portacontenedores con un CMC conocido, que no se puede exceder bajo ningún concepto (puesto que el buque se hundiría en el mar). La carga está distribuida en contenedores, todos del mismo tamaño, pero pudiendo contener diferentes pesos dependiendo del producto que haya en su interior. En el puerto hay **n** contenedores disponibles para ser cargados en buques, cada uno ($1 \leq \mathbf{i} \leq n$) tiene de peso $\boldsymbol{p_i}$ conocido para poder proceder a su carga, sabiendo que la capacidad máxima de carga (CMC) del buque es **C** y que ésta no se puede superar.

Bajo las anteriores condiciones, la aplicación que tiene que desarrollar **EDASoft** debe permitir poder cargar el buque con la máxima carga posible, dependiendo del número de contenedores disponibles en el puerto y respetando su CMC. Este problema se puede formular como un **problema de optimización** de la siguiente manera: en **EDAPort-Almeria** hay un total de **n** contenedores disponibles para ser cargados en un buque con CMC **C**, cada uno con un peso p_i ($1 \leq i \leq$ n), y sea x_i ($1 \leq i \leq$ n) una variable cuyo valor puede ser 0 o 1, $x_i \in \{0, 1\}$, donde $x_i = 0$ significa que el contenedor ***i*** no se cargará en el buque; y $x_i = 1$ indica que sí se carga. Lo que se pretende es asignar valores a x_i para que satisfagan las condiciones de que $\sum_{i=1}^{n} p_i \cdot x_i \leq C$ y $x_i \in \{0, 1\}, 1 \leq i \leq n$. Es decir, el objetivo principal es **maximizar** la carga del buque $\sum_{i=1}^{n} p_i \cdot x_i$, sujeto a las **restricciones** $\sum_{i=1}^{n} p_i \cdot x_i \leq C$ y $x_i \in \{0, 1\}, 1 \leq i \leq n$. La función de optimización es $\sum_{i=1}^{n} p_i \cdot x_i$, y cada conjunto de x_i que satisface las restricciones, es una *solución factible*. Toda solución factible que maximice $\sum_{i=1}^{n} p_i \cdot x_i$ es una *solución óptima*.

Trabajo a desarrollar

Deberá proponer e implementar soluciones con los esquemas algorítmicos de **greedy**, **backtracking** y **branch-and-bound**, según se requiera, a los problemas que se plantean a continuación:

1. (**Greedy**) El buque podrá cargarse por etapas y en cada etapa se cargará un contenedor, decidiendo cuál es el contenedor más apropiado. Para esta decisión podemos utilizar el siguiente criterio greedy: *de los contenedores restantes, seleccionar el de menor peso*. Este orden de selección mantendrá como mínimo el peso total de los contenedores seleccionados y, por tanto, dejará la capacidad máxima para poder cargar más contenedores. Implementar en Java el algoritmo *greedy* que acabamos de describir (**`GreedyContLoading.java`**). Primero seleccionamos el contenedor de menos peso, luego el siguiente de peso más pequeño, y así sucesivamente hasta que se hayan cargado todos los contenedores o no haya suficiente capacidad en el buque para el siguiente.
2. (**Backtracking**) Como ya sabemos, *Backtracking* es una forma sistemática de buscar la solución a un problema, caracterizado por definir un espacio de solución para dicho problema. Este espacio debe incluir al menos una solución (óptima) a dicho problema. Por ejemplo, para el caso del problema de la Mochila 0/1 con *n* objetos, una elección razonable para el espacio de solución es el conjunto de 2^n vectores 0/1 de tamaño n (estructura de árbol binario). Dado que debemos encontrar un subconjunto de contenedores (pesos) con una suma lo más cercana posible a C, utilizamos un espacio de soluciones que puede organizarse como un ***árbol binario***. Para ello, se buscará primero en profundidad (Depth-First) el espacio de soluciones para encontrar la mejor solución. Utilizaremos una función de acotación (bounding function) para evitar la expansión de nodos que posiblemente no puedan conducir a la respuesta deseada. Si *N* es un nodo en el nivel *j+1* del árbol, entonces el camino desde la raíz hasta *N* define los valores para x_i, $1 \leq i \leq$ **j**. Usando estos valores, definimos *pa* (peso actual) como $\sum_{i=1}^{n} p_i \cdot x_i$,. Si *pa* > C, entonces el subárbol con raíz *N* no puede contener una solución factible y podemos utilizar esto como función de acotación. Por otro lado, podríamos aumentar la expansión de nodos para ver si *pa* = C. Si es así, entonces podemos terminar la búsqueda del subconjunto con la suma más cercana a C. Finalmente, consideramos que un nodo será inviable si su valor de *pa* excede C. Para este apartado, se debe de **explicar detalladamente el programa en Java** que se proporciona (**`RecursiveBTContLoading1.java`**) como primera aproximación para resolver el problema para la carga de contenedores en un buque. Analizar también la *complejidad temporal* y *espacial* del algoritmo implementado en dicho programa.

3. (**Backtracking**) Podemos mejorar el rendimiento esperado del método *backtracking* (`rContLoad`) del apartado anterior al no movernos hacia los subárboles de la derecha que posiblemente no contengan mejores soluciones que las mejores encontradas hasta el momento. Sea *N* un nodo en el nivel *i* del árbol del espacio de soluciones. Ninguna hoja en el subárbol con raíz *N* tiene un peso mayor que `weightOfCurrentLoading + remainingWeight` donde $remainingWeight = \sum_{j=i+1}^{n} p_j$ es el peso de los contenedores restantes (**n** es el número total de contenedores). Por lo tanto, cuando `weightOfCurrentLoading + remainingWeight ≤ maxWeightSoFar` no hay necesidad de buscar el subárbol derecho de *N* (recordar que estamos ***maximizando***). Implementar un nuevo programa en Java (**`RecursiveBTContLoading2.java`**) basado en el del anterior apartado, que lleve a cabo dicha mejora y justifíquelo todo adecuadamente según un ejemplo.
4. (**Backtracking**) Para determinar el subconjunto de contenedores que tiene el peso más cercano a la CMC del buque (**solución**), es necesario añadir código al programa del apartado anterior para recordar el mejor subconjunto de contenedores encontrado hasta el momento. Para recordar dicho subconjunto, se puede añadir el parámetro `bestLoading` al método `maxLoading`. `bestLoading` es un array de enteros (`int`) que pueden tomar valores `{0, 1}`, de modo que el contenedor *i* está en el mejor subconjunto si `bestLoading[i] = 1`. Como sugerencia, se podría añadir a la clase dos datos miembro `static`, `currentLoading` y `bestLoadingSoFar`. Ambos datos miembro son arrays de tipo `int`. El array `currentLoading` se utiliza para registrar la ruta desde la raíz del árbol de búsqueda hasta el nodo actual (es decir, guarda los valores x_i en esta ruta), y `bestLoadingSoFar` registra la mejor solución encontrada hasta el momento. Cada vez que se alcanza una hoja con un mejor valor, `bestLoadingSoFar` se actualiza a ese valor de hoja. Los valores de dicho camino identifican los contenedores a cargar. El espacio para el array `currentLoading` se asignaría en el método `maxLoading`, mientras que como hemos dicho antes `bestLoadingSoFar` sería el parámetro `bestLoading`. Implementar un nuevo programa en Java (**`RecursiveBTContLoading3.java`**) basado en el del anterior apartado, que realice dicha mejora (**solución**) y justificándolo todo.
5. (**Backtracking**) La versión recursiva implementada en el apartado anterior se puede mejorar, reduciendo sus necesidades de espacio (memoria). Podemos eliminar el espacio de la pila de recursividad, que es *O(n)*, ya que el array `currentLoading` almacena toda la información necesaria para moverse en el árbol binario del espacio de soluciones. Como ya sabemos, desde cualquier nodo en el árbol binario del espacio de soluciones, nuestro algoritmo realiza una serie de movimientos del hijo izquierdo hasta que no se pueden progresar más. Luego, si se ha alcanzado una hoja, se actualiza la mejor solución, y de lo contrario, intenta pasar al hijo adecuado. Cuando se alcanza un nodo hoja, si no vale la pena realizar un movimiento hacia el hijo derecho, el algoritmo retrocede en el árbol binario hasta un nodo desde el cual se puede realizar un movimiento hacia el hijo derecho. Este nodo tiene la propiedad de que es el nodo más cercano en el camino desde la raíz que tiene `currentLoading[i] = 1`. Si un movimiento hacia el hijo derecho es posible, se realiza y nuevamente intentamos hacer una serie de movimientos hacia el nodo de la izquierda. Si el movimiento al hijo correcto no es posible, retrocedemos al siguiente nodo con `currentLoading[i] = 1`. Este movimiento del algoritmo a través del árbol binario se puede implementar como un **algoritmo iterativo**. A diferencia del código recursivo, esta versión iterativa pasa al hijo derecho antes de comprobar si debería hacerlo, y si el movimiento no debería haberse realizado, el código vuelve atrás. Codificar un nuevo programa en Java (**`IterativeBTContLoading.java`**) que implemente la solución a nuestro problema en su **versión iterativa** y justificándolo todo adecuadamente. Este apartado es **optativo para los grupos de 2 miembros** y **obligatorio para los grupos de 3**.

Destacar que originalmente la práctica tenía 4 apartados más, todos ellos relativos a **Branch-and-Bound** y se han omitido por ahora, con el objetivo de no agobiaros con el desarrollo de esta última práctica. Si posteriormente observamos que se pueden desarrollar algunos de ellos, entonces os los pasaremos como una extensión de esta práctica, pero por el momento, la práctica es solo de **Backtracking**.

Para ello deberá realizar los siguientes apartados:

- **Estudio de la implementación**: Explicar los detalles más importantes de la implementación llevada a cabo, tanto de las estructuras de datos utilizadas para resolver el problema en cuestión, como de los algoritmos implementados. El código debe de estar razonablemente bien documentado (`JavaDoc`).
- **Estudio teórico**: Estudiar los *tiempos de ejecución* (complejidad temporal) y los recursos consuido (complejidad espacial) de los algoritmos ***greedy***, ***backtracking*** y ***branch-and-bound*** implementados, en función del número del número de contenedores a cargar en el buque. Realizar una tabla comparativa de las tres técnicas algorítmicas vistas en esta práctica: *greedy*, *backtracking*, y *branch-and-bound* destacando sus características, ventajas, desventajas, etc.
- **Estudio experimental**: Validación de los algoritmos ***greedy***, ***backtracking*** y ***branch-and-bound*** implementados. Para ello, se podrán utilizar los conjuntos de datos disponibles en la siguiente Web: https://people.sc.fsu.edu/~jburkardt/datasets/bin_packing/bin_packing.html. Podemos ver que existen 4 conjuntos, y en ellos se pueden observar datos relativos a: capacidad del buque y pesos de los contenedores (ignorar los archivos relativos a 'the optimal assignment of weights'). Para ello, se deberán comprobar el correcto funcionamiento y comparar los tiempos de ejecución de los algoritmos implementados. Se contrastarán los resultados teóricos y los experimentales, comprobando si los experimentales confirman los teóricos previamente analizados. Se justificarán los experimentos realizados, y en caso de discrepancia entre la teoría y los experimentos se debe intentar buscar una explicación razonada. Además, se generarán **datos aleatorios** (de peso de contenedores y capacidad del buque), en concordancia con el formato de los disponibles en la Web, para valores de **n** relativamente grandes, comprobando si se aprecia alguna variación en los tiempos de ejecución de los algoritmos implementados en esta práctica (sobre todo para extraer conclusiones en función de su tiempo de ejecución en función de su *complejidad temporal*).

Entregas

Se ha de entregar, en fecha, en el repositorio GitHub (mismo repositorio para todas las prácticas de EDA II) con toda la documentación y el código fuente requerido en la práctica:

- En dicho repositorio, en la carpeta de fuentes `src/main/java`, crear un nuevo paquete llamado `org.eda2.practica04`, para el **código fuente** de la práctica. Además, en esta carpeta deben de incluirse los archivos de datos de la Web y los generados de forma sintética.
- En la carpeta `docs`, dedicada a la **documentación**, crear una subcarpeta `practica04` para guardar toda la documentación (documento en `pdf` y los fuentes utilizados para su creación (por ejemplo, `.docx`)).
- **Memoria** que explique todo lo que habéis realizado en la práctica. La memoria deberá tener el formato que se indica a continuación. Si se desea, también se podrá realizar una presentación de la práctica.

- **Código fuente** de la aplicación, desarrollada en JAVA o en C++, que resuelva todo lo planteado en la práctica.
- **Juegos de prueba** que consideréis oportunos incluir para justificar que todo funciona correctamente (fundamentalmente para los casos de los datos que hay disponibles en la Web). En este caso los juegos de prueba deben estar en la carpeta de fuentes llamada `src/test/java` y dentro de ella en un paquete llamado `org.eda2.practica04`.

La memoria de práctica a entregar debe ser breve, clara y estar bien escrita. Ésta debe incluir las siguientes secciones:

- Una breve **introducción** con un estudio teórico de los métodos algorítmicos utilizados en esta práctica (***backtracking*** y ***branch-and-bound***).
- Una sección para cada uno de los **apartados propuestos** a desarrollar en esta práctica (estudio de la implementación, estudio teórico y estudio experimental). Hemos de remarcar que deben incluirse los apartados en el mismo orden en el que se han expuesto.
- Se incluirá también un **anexo** con el diseño del código implementado, con diagramas de clases y cualquier otro diagrama que estiméis necesario incluir, no introducir ningún código en este anexo. Además, añadir en esta sección una lista de los archivos fuente y una breve descripción del contenido de cada uno.
- Es importante incluir siempre las **fuentes bibliográficas** utilizadas (web, libros, artículos, etc.) y hacer referencia a ellas en el documento.

La **memoria de la práctica** a entregar debe ser breve, clara y estar bien escrita, adaptándose en lo posible a la norma UNE 157001:2014 "Criterios generales para la elaboración de proyectos". Ésta debe incluir las siguientes secciones:

- Un apartado **Objeto** con un estudio teórico de los métodos algorítmicos utilizados en esta práctica (*backtracking* y *branch-and-bound*). Además, en esta sección deberá exponerse claramente ***qué miembro* del grupo ha actuado como *líder* y *qué tareas ha realizado cada miembro del grupo***. Es muy importante que todos los miembros dominen la práctica en su conjunto. Se puede incorporar la hoja de datos indicada para tal fin (`PR4_Tareas_a_repartir.xlsx`).
- Un apartado **Antecedentes** en el que se explique el motivo que lleva a realizar esta memoria y se enumeran los aspectos necesarios, en su caso, de las alternativas contempladas y la solución final adoptada.
- Una sección para cada uno de **apartados propuestos** a desarrollar en esta práctica (estudio de la implementación, estudio teórico y estudio experimental). Para el **estudio experimental**, el correcto funcionamiento de los algoritmos debe de justificarse con la correspondiente captura de pantalla en la que se pueda apreciar la salida correcta. Hemos de remarcar que deben incluirse los apartados en el mismo orden en el que se han expuesto. El apartado **estudio experimental** de la memoria recogerá los resultados finales de lo realizado.
- Se incluirá también un **anexo** con el diseño del código implementado, con diagramas de clases y cualquier otro diagrama que estiméis necesario incluir, **no introducir ningún código en este anexo**. Además, añadir en esta sección una lista de los archivos fuente que componen la práctica y una breve descripción del contenido de cada uno.

- En el apartado de **estudio experimental** o en un **anexo**, se expondrán los cálculos realizados en el estudio experimental, con la explicación del método seguido para la toma de datos y gráficos con el resultado obtenido (datos de muestras y ecuación seleccionada). Cuando se elija una curva para explicar el orden de un algoritmo (por ejemplo, O(n)), se deberá añadir el estadístico de la varianza explicada R^2, de forma que este valor (que varía entre 0 y 1) debería ser superior a 0.95 para que se acepte la solución propuesta. Valores por debajo 0.90 indican que el proceso de toma de muestras o el de elaboración de la hipótesis, ha sido incorrecto y debería llevar a su revisión. Cada muestra debería ser una media de varias medidas (al menos 10) de tiempo de ejecución y el tiempo medido debería ser, al menos, cercano al segundo para evitar interferencias del sistema operativo.
- Es importante incluir siempre las **fuentes bibliográficas** utilizadas (web, libros, artículos, etc.) y hacer referencia a ellas en el documento.

Evaluación

Cada apartado se evaluará independientemente, aunque es condición necesaria para aprobar la práctica que los programas implementados funcionen correctamente.

- La implementación junto con la documentación del código se valorará sobre un 40%
- El estudio de la implementación se valorará sobre un 10%
- El estudio teórico se valorará sobre un 15%
- El estudio experimental se valorará sobre un 35%

Se penalizará no entregar el apartado de introducción teórico o una mala presentación de la memoria.

Se podrá requerir la defensa del código y de la memoria por parte de profesor.

Fecha de entrega: 6 de Junio de 2024