Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

Algoritmos avanzados

Laboratorio 3 –
Ramificación y acotamiento

Christofer Rodríguez

Profesor: Cristián Sepúlveda S.

Ayudante: César Rivera M.

Santiago – Chile
1-2021

Tabla de contenidos

Índice de figuras

Índice de tablas

1. INTRODUCCIÓN

La seguridad informática ha sido unos de los temas más importantes por mucho tiempo, con el fin de fortalecer la seguridad del cifrado y descifrado de claves Kasahara y Murakami propusieron una variación al antiguo esquema, capaz de proteger estas claves de distintos tipos de ataques. Esta variación del esquema consiste en un conjunto de números enteros con una ponderación correspondiente, con esto en mente es necesario implementar un programa en el lenguaje de programación C, que haciendo uso de algoritmos de ramificación y acotamiento, sea capaz de determinar la mayor suma que se puede obtener de un subconjunto de números que a su vez no superen un límite en la suma total de sus ponderaciones.

Los objetivos de este informe son explicar el método utilizado para resolver este problema, describir los experimentos realizados para la obtención de datos, para finalmente analizar los resultados obtenidos, identificando los principales hallazgos y limitaciones de la solución propuesta.

Esto se llevará a cabo, primeramente, describiendo la metodología utilizada, explicando en que consiste el tipo de algoritmo utilizado, el proceso de implementación y los resultados obtenidos en los distintos experimentos realizados; luego se analizarán los resultados obtenidos en la sección de discusión, en esta sección se expondrán los principales hallazgos, falencias de la implementación y se comparará con los resultados obtenidos en el anterior laboratorio, finalmente sugerirán mejoras para futuras investigaciones e implementaciones.

2. MÉTODO

Para comprender la implementación realizada, es necesario primero comprender en que consiste un algoritmo ramificación y acotamiento; este tipo de enfoque es una variante de los algoritmos de backtracking. Al igual que en backtracking se suele modelar el problema en un árbol de soluciones, donde cada uno de los nodos que componen el árbol, puede ser una posible solución generada a partir del nodo anterior. La diferencia que tiene este enfoque con backtracking, es la realización de la llamada poda, con esto se refiere a descartar las ramificaciones en donde las soluciones que pueden ser obtenidas en ellas, ya no son óptimas. La ventaja de este tipo de algoritmo es que, en comparación a la técnica de enumeración exhaustiva, posee igual calidad de respuesta y menor tiempo de respuesta, a su vez si la comparamos con un algoritmo voraz, puede tardar más tiempo en entregar una respuesta, pero esta posee mejor calidad de respuesta, ya que siempre entrega el óptimo.[1]

Para solucionar el problema propuesto con un enfoque de ramificación y acotamiento en el lenguaje de programación C, se desarrolló

2.1. Solución problema

Para esta implementación se decidió hacer uso de estructuras para representar los elementos con sus respectivos valores, ponderaciones, además un valor que representa la relación entre su valor con respecto a su ponderación, este valor se obtuvo dividiendo ambos datos respectivamente.

Ya que los datos para este problema son ingresados mediante un archivo de texto plano con cierta estructura, se implementó un algoritmo que, tras recibir el nombre del archivo de entrada, abre dicho archivo en modo lectura, y a la vez que este era leído, la información contenida en él se almaceno en la estructura que representa cada elemento. Luego de generar el elemento con la información correspondiente, este se agrega a un arreglo que, tras haber leído la totalidad del archivo, es retornado con todos los elementos en él.

Una vez leído el archivo de entrada, se comprueba que no haya existido un error al momento de leer dicho archivo. Si se comprueba que no existió un problema, se procede a ordenar los elementos de mayor a menor con respecto a su valor por ponderación, esto se realizó utilizando el ordenamiento de merge sort [2].

En cuanto a la resolución del problema, se decidió modelar este como un árbol de decisiones, en el cual, comenzando desde la raíz, el hijo izquierdo corresponde a considerar el elemento actual dentro del conjunto solución, a su vez el hijo derecho corresponde a no considerar dicho elemento dentro del conjunto solución. Debido a que no es necesario conocer los elementos que componen la solución, sino que solamente la suma total de estos, se tomó la

decisión de almacenar los nodos en una lista simplemente enlazada, en vez de construir un árbol, por esto, se construyó una estructura llamada **nodo** la cual representará cada nodo perteneciente al árbol, esta contendrá el nivel del nodo dentro del árbol, su valor, el límite restante de ponderación, su cota y un puntero al siguiente nodo.

Como se nombró anteriormente, al describir la composición de la estructura **nodo**, este posee una variable llamada cota, esta representa el máximo beneficio que podríamos obtener de la ramificación generada por este nodo, esta cota se obtiene resolviendo el subproblema "de manera relajada", esto hace referencia a que no se consideran ciertas restricciones, siendo esta el que la solución debe ser entera. Este cálculo se realiza con una función construida mediante la modificación de un algoritmo con enfoque voraz construido en el laboratorio numero 2; para resolver el problema de manera relajada, se considera que los elementos pueden ser divididos, de manera de agregar los elementos que posean un mayor beneficio en relación con su beneficio por peso y siempre utilizar la totalidad de la ponderación limite.

Después de haber ordenado los elementos, se entregan a la función encargada de generar el árbol mientras busca la solución del problema. Se comienza generando la raíz del árbol e ingresándola a la lista enlazada que contendrá los elementos del árbol y estableciendo una variable que almacenará la mayor solución encontrada, de esta manera la podremos utilizar para saber que nodos siguen siendo posibles óptimos. Luego dentro de un ciclo se busca el índice del nodo que posea la mayor cota, de esta manera siempre trabajaremos con los nodos más prometedores y después este es extraído de la lista.

Extraído el nodo de la lista, se verifica si al conjunto solución que posee este nodo, se pueden seguir agregando elementos o si este ya ha considerado a todos los elementos para la generación de su conjunto solución, si este es el caso no se puede seguir trabajando con este nodo, por lo que es eliminado, liberando la memoria utilizada por este. En el caso de que se pueda seguir trabajando con el nodo, se calcula si introduciendo el siguiente elemento del conjunto de candidatos dentro de la solución es posible, si se puede, se calcula la cota del nodo obtenido al incluir este elemento (hijo izquierdo), para esto se suma el valor del nodo padre, más el valor del elemento incluido y el máximo beneficio que se puede obtener con el resto de los elementos, aquí nos ayudará el nivel del nodo, ya que este representa desde donde se comienza a considerar los elementos de la lista de candidatos al calcular la cota.

Si la cota del nuevo nodo es mayor, a la mayor suma de valores encontrada hasta el momento, se generará el nodo y se almacenará dentro de la lista enlazada, si además el valor de este nuevo nodo es mayor que el mayor encontrado hasta este momento, este lo reemplazara. En el caso de que de que la cota calculada sea menor a la mayor solución encontrada, no se generará el nodo, ya que no podrá obtener una mejor solución que la que ya ha sido encontrada a través de esta ramificación.

Ya sea se haya generado el hijo izquierdo o no, se calculará la cota del hijo derecho, la cual no considera el elemento actual dentro del conjunto solución, al igual que con el hijo izquierdo, se comprueba si la cota obtenida es mayor a la mayor solución encontrada, si no lo es, no se genera y almacena el nodo; en caso de que, si sea mayor, se genera y se almacena. A diferencia de cuando se genera el hijo izquierdo, en esta ocasión no se comprueba si el

valor obtenido en este nuevo nodo es mayor a la mayor solución encontrada, ya que el valor del nuevo nodo es igual al valor del nodo padre y este ya habrá sido comprobado.

Al final de la iteración se elimina el nodo con el que se estaba trabajando y se libera la memoria utilizada por este. Las maneras de obtener una respuesta optima son 2, la primera consiste en que el valor de un nuevo nodo generado es igual a su cota, esto ya que como estamos seleccionando siempre el nodo con mayor cota y que a medida que se desciende en el árbol de decisiones, la cota solamente puede mantenerse o disminuir, que el valor sea igual a la cota significa que se obtuvo el máximo valor posible de la ramificación y a su vez, del árbol, por lo que se retorna este valor como respuesta. La segunda forma de obtener una respuesta es que sea hayan revisado todos los nodos de la lista enlazada y esta se encuentre vacía, en este caso se retornará el valor de la mayor solución encontrada como respuesta al problema.

El código de esta solución fue analizado contando la cantidad de instrucciones a ejecutar para obtener el tiempo de ejecución teórico y el orden de complejidad respectivamente:

- $T(n) = 1 + 10 * 2^n + n * 2^n + n^3 + n^3 * 2^n$
- $O(2^n)$.

La ecuación del orden de complejidad se utilizó para representar el aumento en el tiempo de ejecución del algoritmo en el siguiente gráfico:

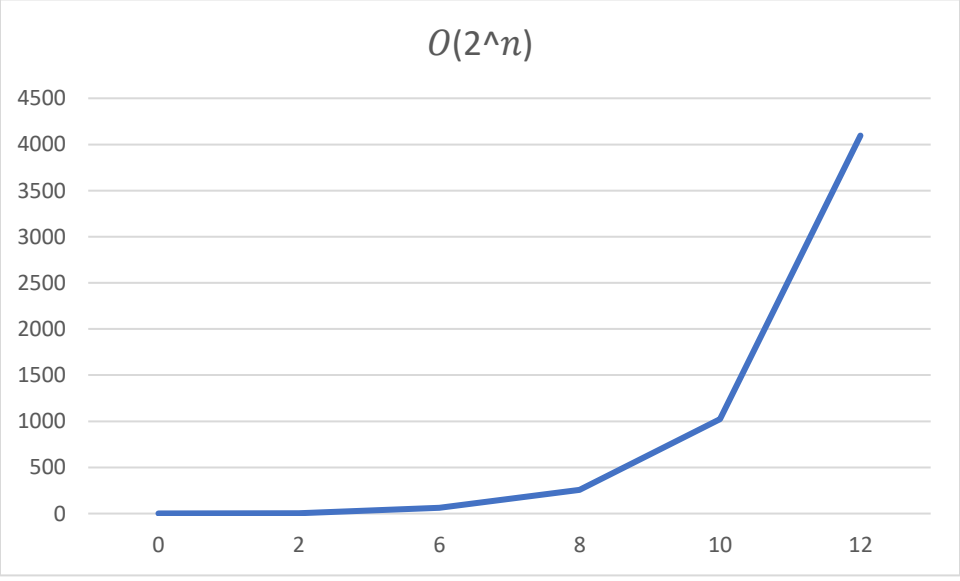

Ilustración 1: Gráfico complejidad algorítmica

Además, para contrastar el tiempo teórico con el tiempo real de ejecución, se hizo uso de la librería **time.h** para la toma del tiempo de ejecución del programa con las entradas.

Tabla 1: Tamaño entrada vs tiempo de ejecución

Tamaño de entrada	Tiempo de ejecución (seg)
10	0
20	0
30	0
40	0
1000	0
10000	0

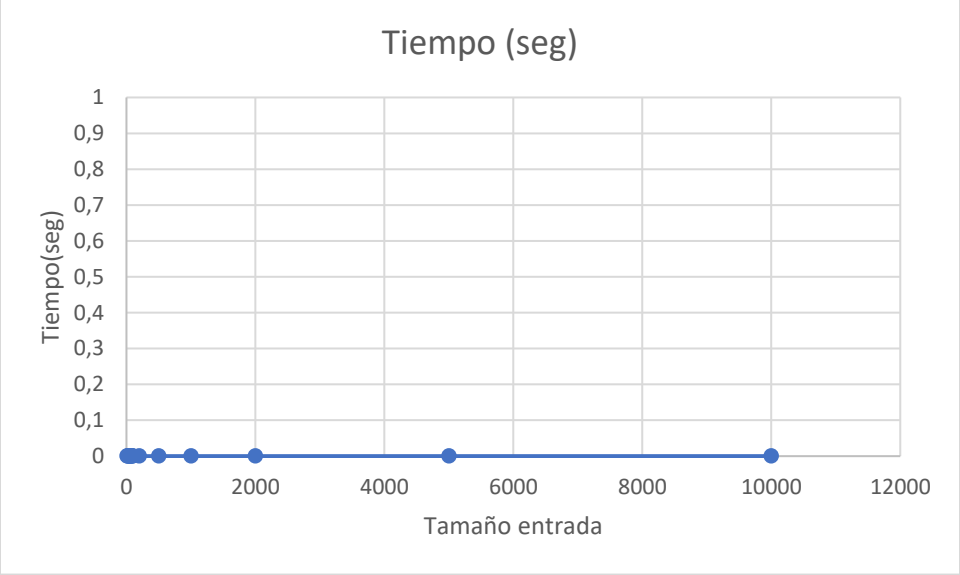

Ilustración 2: Tiempo de ejecución real

(*) Es importante señalar que los experimentos realizados con ambos programas se realizaron en un computador común con sistema operativo Windows 10 , procesador Intel Core i3-8100 de 3.60 GHz y 16 GB de memoria RAM. Los resultados pueden variar en un computador, entorno o situación diferente.

3. DISCUSIÓN

3.1. Análisis solución

Luego de analizar los resultados obtenidos de los experimentos realizados con el programa construido, se puede observar que la aplicación del método de ramificación y acotamiento nos entrega una solución al problema de manera instantánea para cada una de las entradas probadas. Si bien mediante la Ilustración 1 y la Ilustración 2, podemos notar que existe una diferencia entre el tiempo teórico de ejecución y el tiempo real, esto se debe a que el tiempo de respuesta del algoritmo de ramificación y acotamiento para este caso es increíblemente bajo, incluso para entradas grandes, quizás si se hubiera utilizado un método que entregará un tiempo de ejecución más preciso, la forma de las curvas sería semejante. Si comparamos los resultados obtenidos con este enfoque y los comparamos con los resultados obtenidos en la experiencia anterior, en donde se resolvió el mismo problema, pero mediante la aplicación de enumeración exhaustiva.

Tabla 2: Tamaño de entrada vs tiempo de ejecución, enumeración exhaustiva

Tamaño de entrada	Tiempo de ejecución (seg)
10	0
20	0,225
30	138,716
40	3600+

***Estos resultados de la experiencia anterior fueron medidos utilizando la misma máquina utilizada para probar los programas de este trabajo.**

Si comparamos los resultados obtenidos mediante ramificación y acotamiento (Tabla 1) con los obtenidos utilizando enumeración exhaustiva (Tabla 2), se puede ver la gran mejora en el tiempo de respuesta, incluso para los tamaños de entrada más grandes, para los que el método de enumeración exhaustiva no fue capaz de entregar una respuesta en el tiempo de prueba designado. A su vez, si comparamos los resultados obtenidos en este laboratorio con los obtenidos mediante el enfoque voraz, el tiempo de respuesta es el mismo a primera vista, esto ya que, al medir el tiempo de respuesta en segundos, todos los resultados fueron de 0 segundos, aunque el enfoque voraz debería tener un menor tiempo de respuesta que ramificación y acotamiento. Además, donde mejora este método frente al enfoque voraz, es que este nos asegura el óptima para todos los casos, a diferencia del enfoque voraz que nos entregaba el óptimo solo en algunas ocasiones como podemos observar en la Ilustración 3.

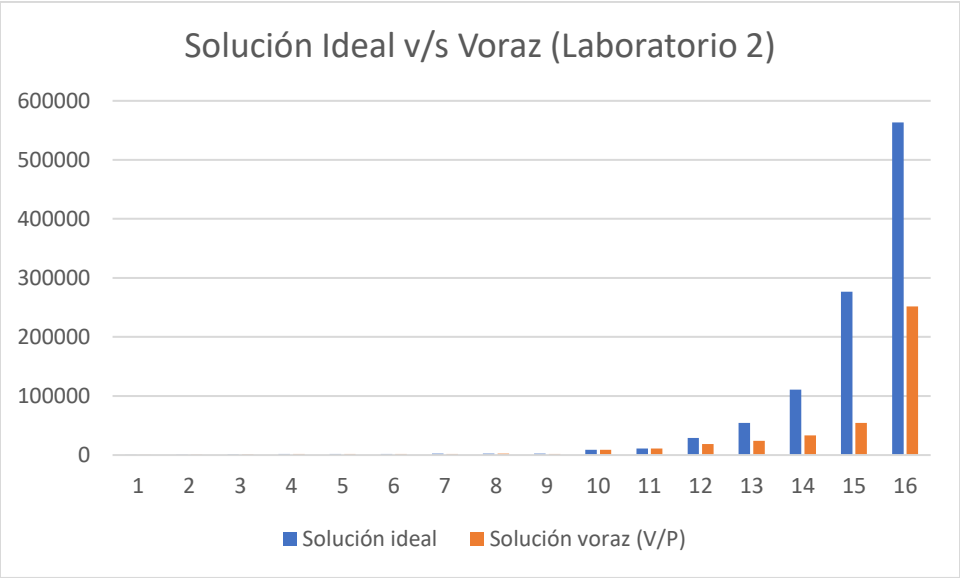

Ilustración 3: Solución ideal vs solución voraz, laboratorio 2

Si bien el método de ramificación y acotamiento nos asegura siempre el óptimo, como falencia de la implementación construida se puede destacar que no se logró obtener el mismo óptimo indicado en los archivos de prueba para los tamaños más grandes de entrada.

Tabla 3: Solución óptima vs obtenida, ramificación y acotamiento

Tamaño	Ideal	Obtenida
10	295	295
20	1024	1024
30	1500	1500
40	1900	1900
50	1896	1896
60	2088	2088
70	2200	2200
80	2185	2185
90	2200	2200
100	9147	9147
200	11238	11238
500	28857	18582
1000	54503	24091
2000	110625	33449
5000	276457	54333
10000	563647	75970

Como se puede observar en la Tabla 3, el óptimo obtenido desde los tamaños 10 al 200, son iguales a los indicados en los archivos de prueba, pero entre el 500 y 10000, el óptimo difiere. Se intentó solucionar este problema durante el desarrollo, pero no se logró, ya que no se pudo identificar el origen de este error, como la cota nos indica el mayor beneficio que podemos obtener de la ramificación de un nodo, las cotas de los hijos derecho e izquierdo de la raíz del árbol de decisión, debe ser igual o mayor al optimo indicado en el archivo de prueba, pero el obtenido era menor a este, por lo que no era un error generado en la ramificación del árbol, dejándonos solo la alternativa de que el origen fuera el cálculo de la cota de un nodo, aunque si este fuera el caso, también deberían haberse presentado errores con entradas pequeñas, lo cual no ocurrió.

En cuanto al mejoramiento de la implementación, se puede indicar que, a pesar de no poder disminuir el orden de complejidad del algoritmo, ya que en el peor de los casos se tendrá que comprobar todas las soluciones posibles, si se puede cambiar la manera de almacenar los datos, esto debido a que se decidió utilizar una lista simplemente enlazada por su simpleza en construcción, pero quizás una estructura de datos diferente nos pueda permitir reducir la cantidad de instrucciones ejecutadas. Además, si se realiza la "poda" de nodos de una manera diferente, se podría evitar revisar una elevada cantidad de nodos. Aunque en la implementación se realizo la poda de manera indirecta, no generando los nodos que tuvieran una cota menor a la mayor solución encontrada hasta el momento, pueden existir nodos que se encuentren anteriormente almacenados en la lista, los cuales ya no sean óptimos, para mejorar esto, se debería realizar una comprobación de los nodos almacenados cada vez que encontremos una nueva solución mayor y eliminar los nodos que ya no sean óptimos.

4. CONCLUSIÓN

A pesar de los errores en la solución para entradas grandes, se logró comprobar que el comportamiento de la curva que representa el tiempo de ejecución real frente a diferentes tamaños de entrada se asemejaba bastante a la curva de la complejidad algorítmica obtenida del tiempo de ejecución teórico, además que se consiguió conocer las ventajas y limitaciones que posee el método de ramificación y acotamiento de manera aislada, y las que posee frente a algoritmos de enumeración exhaustiva. Como principal aprendizaje se puede destacar la vital importancia de considerar la precisión necesaria de la respuesta, el tiempo empleado en obtener esta y la complejidad de implementar el programa al momento de elegir el enfoque más adecuado para enfrentar la problemática. Todo lo aprendido y obtenido en la realización de este laboratorio, será beneficioso para la realización de futuras implementaciones a desarrollar en distintos campos de estudio.

5. REFERENCIAS

1. Berzal, F. (2010). 5 exploración de grafos. (Recuperado el 25/08/2021) https://elvex.ugr.es/decsai/algorithms/slides/5%20Branch%20and%20Bound.pdf

2. GeeksforGeeks(2021). Merge Sort (Recuperado el 06/06/2021) https://www.geeksforgeeks.org/merge-sort/