Trabajo Práctico 2: Problema de Empaquetamiento

Teoría de Algorítmos I

Cátedra Buchwald

Mayo, 2023

Integrantes

Nombre	**Padrón**
Bedoya, Gabriel	107602
Berenguel, Rafael	108225

Índice

1. Problema de Empaquetamiento

Dado un conjunto de n objetos cuyos tamaños son $\{T_1, T_2, \ldots, T_n\}$, con $T_i \in (0, 1]$, se debe empaquetarlos usando la mínima cantidad de envases de capacidad 1.

Consigna:

1. Demostrar que el problema de empaquetamiento es NP-Completo.
2. Programar un algoritmo por Backtracking/Fuerza Bruta que busque la solución exacta del problema. Indicar la complejidad del mismo. Realizar mediciones del tiempo de ejecución, y realizar gráficos en función de n.
3. Considerar el siguiente algoritmo: Se abre el primer envase y se empaqueta el primer objeto, luego por cada uno de los objetos restantes se prueba si cabe en el envase actual que está abierto. Si es así, se lo agrega a dicho envase, y se sigue con el siguiente objeto. Si no entra, se cierra el envase actual, se abre uno nuevo que pasa a ser el envase actual, se empaqueta el objeto y se prosigue con el siguiente.

   Este algoritmo sirve como una aproximación para resolver el problema de empaquetamiento. Implementar dicho algoritmo, analizar su complejidad, y analizar cuán buena aproximación es. Para esto, considerar lo siguiente: Sea I una instancia cualquiera del problema de empaquetamiento, y $z(I)$ una solución óptima para dicha instancia, y sea $A(I)$ la solución aproximada, se define $\frac{A(I)}{z(I)} \leq r(A)$ para todas las instancias posibles. Calcular $r(A)$ para el algoritmo dado, demostrando que la cota está bien calculada. Realizar mediciones utilizando el algoritmo exacto y la aproximación, con el objetivo de verificar dicha relación.
4. [Opcional] Implementar alguna otra aproximación (u algoritmo greedy) que les parezca de interés. Comparar sus resultados con los dados por la aproximación del punto 3. Indicar y justificar su complejidad.

Se recomienda realizar varias ejecuciones con distintos conjuntos de datos del mismo tamaño y promediar los tiempos medidos para obtener un punto a graficar. Repetir para valores de n crecientes hasta valores que sean manejables con el hardware donde se realiza la prueba.

1.1. Demostración NP-Completo

Para demostrar que el problema propuesto es NP-Completo lo primero que podemos hacer es plantearlo de manera distinta. El problema del que se nos habla es un problema de optimización, sin embargo para probar que es NP-Completo lo correcto sería hacerlo sobre un problema de decisión. Por lo que, en lo subsiguiente, el problema que se intentará probar NP-Completo será el de

> ¿Se pueden poner n objetos cuyos tamaños son $\{T_1, T_2, \ldots, T_n\}$ con $T_i \in (0, 1]$ en, como mucho, K envases de capacidad 1?

Teniendo el problema planteado de esta manera podemos probar que es NP-Completo.

Sabemos que un problema **X** es NP-Completo si y solo si se cumple que:

1. $X \in \text{NP}$
2. $\forall Y \in \text{NP}, Y \leq_p X$

Sin embargo, debido a que ya son conocidos muchos otros problemas NP-Completos, podemos hacer uso de una propiedad que dice que:

> Si Y es un problema NP-Completo, y X es un problema en NP, con la propiedad de que $Y \leq_p X$, entonces X es NP-Completo.

Sabiendo esto, las condiciones que tenemos que cumplir para demostrar que nuestro problema es NP-Completo son:

1. Nuestro problema debe pertenecer a NP.
2. Un problema que se sabe NP-Completo puede ser reducido a nuestro problema.

1.1.1. ¿Es NP nuestro problema?

Para probar que nuestro problema es NP completo se debe poder decidir en complejidad polinomial si una solución es válida o no.

La posible solución sería una asignación de objetos a los K envases. Para verificar si dicha solución es válida, debemos comprobar:

1. El tamaño total de cada uno de los K-envases no supere el tope 1.
2. Los elementos que se tienen que poner en los envases están presentes una sola vez en un solo envase.

Ambas verificaciones pueden realizarse en tiempo polinómico, al poder recorrerse la asignación y calcular las sumas necesarias. Por lo tanto, nuestro problema está en NP.

1.1.2. Reduciendo un problema NP completo a nuestro problema

Reducir un problema NP-Completo a nuestro problema consiste en tomar nuestro problema y moldearlo de una forma en la que pueda resolver cualquier caso del problema NP-Completo, demostrando que nuestro problema inicial es más grande y más complejo, pues el problema conocido NP-Completo se puede resolver como un caso de este.

Para nuestro caso el problema que elegimos reducir al problema de empaquetamiento es el problema de partición, uno de los problema NP-Completos más conocidos.

El problema de partición consiste en poder decir si, teniendo una lista de distintos números cuya suma total es A, podemos dividirla en dos subsets que al sumarse nos den cada uno el mismo número, el cual tendrá que ser la mitad de A.

Por ejemplo, si tenemos la lista [3, 4, 6, 4, 2, 7], la suma de los elementos de esta nos da 26 y la podemos dividir en los subsets [3, 4, 6] y [4, 2, 7], ambos teniendo como la suma de sus elementos el número 13, con lo cual el problema de partición nos diría que existe una solución.

Por otra parte si tuviéramos la lista [3, 4, 6, 8, 4, 2, 7, 5], la suma de los elementos de esta nos da 39 y no hay manera de poder separar estos números de manera de obtener dos subsets de igual suma de elementos.

Para poder reducir este problema a nuestro problema de empaquetamiento en su versión de decisión, lo que podemos hacer es preguntarnos si podríamos meter todos los elementos de la lista en, como mucho, dos envases, con cada envase teniendo como capacidad máxima la mitad de la suma de todos los elementos de la lista, pues si pudimos separarlos en dos envases entonces se separaron en dos grupos cuya suma nos da el mismo número, y si requieren de más envases entonces esto no era posible.

Sin embargo, esta solución no tiene en cuenta una característica más de nuestro problema, y es que la capacidad de los envases no es variable, sino que está fija en 1. Afortunadamente tenemos una manera de poder solventar esto. Podemos reducir el tamaño de los números manteniendo la proporción entre estos, para que la suma total de como resultado 2 y en cada envase solo tengamos que rellenar hasta el número 1. Esto lo podemos lograr dividiendo cada elemento de la lista por la mitad de la suma total.

Volvamos al ejemplo de antes, en donde el problema de partición tenía solución. Tenemos la siguiente lista de elementos $[e_0, e_1, \ldots, e_n]$:

$$[3, 4, 6, 4, 2, 7]$$

Si llamamos S a la sumatoria de todos los elementos de la lista, $\sum_{i=0}^{n} e_i$:

$$S = 3 + 4 + 6 + 4 + 2 + 7 = 26$$

Dividimos a todos los elementos de la lista por $\frac{S}{2}$

$$\frac{[3, 4, 6, 4, 2, 7]}{\frac{26}{2}}$$

$$\frac{[3, 4, 6, 4, 2, 7]}{13}$$

$$[0{,}23, 0{,}31, 0{,}46, 0{,}31, 0{,}15, 0{,}54]$$

Ahora al preguntarnos "¿Se puede poner todos estos elementos en 2 envases de capacidad 1?" si la respuesta es **sí**, entonces se pudieron dividir bien, y si es **no** entonces no, pues no se pueden usar menos de dos envases al ser suma de todos los elementos 2, y si se usan 3 o más entonces no encajan bien los números para ser separados.

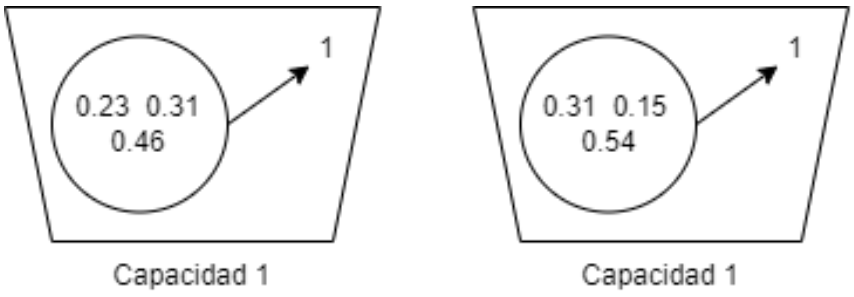

Figura 1: Elementos de una lista dividiéndose en dos vasos de capacidad 1

En el caso que presentamos que no daría un resultado positivo se daría por ejemplo que

$$[3, 4, 6, 8, 4, 2, 7, 5]$$

$$S = 39$$

$$\frac{[3, 4, 6, 8, 4, 2, 7, 5]}{19{,}5}$$

$$[0{,}15, 0{,}21, 0{,}31, 0{,}41, 0{,}21, 0{,}10, 0{,}36, 0{,}25]$$

Cuando nuestro programa se pregunte si estos elementos se pueden acomodar en 2 envases de capacidad 1, verá que esto no es posible.

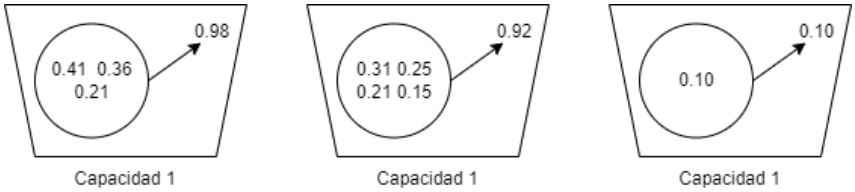

Figura 2: Elementos de una lista dividiéndose en tres vasos de capacidad 1

Por lo que, efectivamente, podemos resolver el problema de partición utilizando nuestro problema de empaquetamiento, probando que un problema NP-Completo puede ser reducido a este.

Con este punto y el anterior, se prueba que el problema de empaquetamiento es NP-Completo.

1.2. Algoritmo con Backtracking

Para realizar un algoritmo por backtracking o fuerza bruta que resuelva nuestro problema vale la pena recordar a qué se refiere este algoritmo.

Un algoritmo de fuerza bruta es un algoritmo que va a evaluar todas las soluciones posibles de un problema para encontrar la adecuada. Suelen ser muy lentos y costosos computacionalmente.

Los algoritmos de backtracking tienen un funcionamiento similar a los de fuerza bruta, pero con la diferencia de que mientras se está avanzando por el árbol de posibles soluciones, si sabemos que en la rama en la que estamos parados no vamos a encontrar una mejor solucione, directamente paramos de probar esos casos, no porque no los tengamos en cuenta, sino porque ya sabemos que no resolverán nuestro problema por lo que ya vimos.

1.2.1. Explicación del algoritmo

En nuestro caso logramos hacer un algoritmo de backtracking que resolviera el problema de empaquetamiento de manera óptima.

La cosa que más costó pensar a la hora de definir lo que sería un algoritmo que probara todas las soluciones era pensar cómo se tendrían que ver estas. Intentamos hacer algoritmos que probaran poner todos los objetos donde les fuera posible, en todos lo ordenes distintos que se pudiera. Este tipo de algoritmos que exhaustivamente buscaban entre todas las soluciones posibles, aunque eventualmente llegan a la solución óptima, tienen un problema muy grande de complejidad, y es que se toma como soluciones diferentes a un orden diferente de paquetes, cuando no lo son.

Expliquemos esta idea más a fondo. Digamos que tenemos los objetos

$$[a, b, c, d, e, f, g]$$

Si nuestro algoritmo separara a los objetos en

$$[a, b, c][d, e, f][g]$$

¿Sería eso distinto de si los separara en lo siguiente?

$$[d, e, f][a, b, c][g]$$

La respuesta es que no, pues los mismos elementos están empaquetados con los mismos elementos y la cantidad de paquetes es la misma, sin embargo, a la hora de hacer un algoritmo de fuerza bruta, es muy común poder caer en el error de tener un algoritmo que considere estas como soluciones distintas del problema y las explore de igual forma, lo que lleva a un algoritmo mucho más costoso.

Dicho todo esto presentamos el funcionamiento de nuestro algoritmo.

Implementamos un algoritmo que busca todas las combinaciones compatibles de todos los empaques posibles de manera recursiva.

La manera en la que esto se logra es la siguiente:

Primero creamos una variable en la que almacenamos la cantidad mínima de paquetes para actualizarla durante la ejecución de nuestro programa.

Luego llamamos a una función auxiliar que será la recursiva. Esta función va a recibir la lista de objetos, la posición del siguiente objeto a guardar en alguno de los paquetes, el estado actual de los paquetes y la cantidad mínima de paquetes para una solución encontrada hasta el momento.

Dentro de esta función lo que se va a hacer es iterar la lista de paquetes y en cada uno hacer lo siguiente:

- Evaluamos si el objeto en la posición de la lista de objetos que estamos evaluando ahora mismo podría ser guardado en el empaque que estamos evaluando sin sobrepasar el límite.
- Si se puede guardar el objeto, entonces:
  - Se lo agrega al paquete
  - Se llama a la función recursiva aumentando en 1 la posición del objeto que estamos evaluando en los empaques, la cual devuelve la cantidad mínima de las soluciones que encontró en esa rama.
  - Se borra el el elemento del paquete para continuar con las iteraciones, habiendo actualizado la mínima cantidad de paquetes encontrada.

Al terminar las iteraciones por los paquetes disponibles, se crea un nuevo paquete en donde se pone el objeto actual solo y se vuelve a llamar a la función, para que en los siguientes objetos estos puedan también cubrir las opciones en las que están empaquetados solos con nuestro objeto actual. Al finalizar este llamado se borra el nuevo paquete.

Para apreciar mejor la manera en la que nuestro algoritmo cubre todos los casos posibles, miremos este diagrama en el que para transmitir la idea principal se actuará como si los empaques no tuvieran un límite de capacidad.

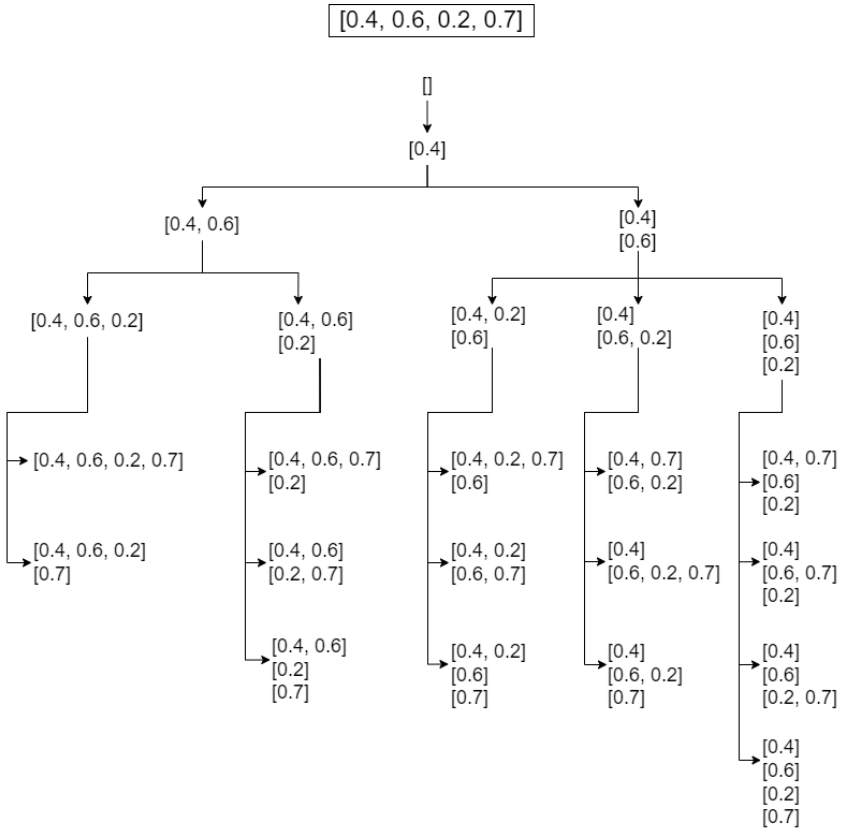

Figura 3: Algoritmo de fuerza bruta si los empaques no tienen límites

En la imagen anterior podemos ver como trabajaría nuestro algoritmo si no se detuviera en ninguna rama ni tuviera empaques con límite de capacidad.

Podemos apreciar como en cada hoja de este árbol tenemos diferentes formas de empaquetar nuestros objetos y todas son combinaciones distintas. Ninguna de las hojas es la de otra rama pero en diferente orden, asegurándonos de estar explorando cada opción posible solo una vez.

Sin embargo, explorar absolutamente todas las opciones no es muy eficiente, pues hay maneras de saber cuando no vale la pena continuar por una rama. Al agregar estas caracterísitcas a nuestro algoritmo este se vuelve uno de **backtracking**.

En primer lugar, nuestro problema tiene un límite de capacidad, por lo que en el momento que crear una nueva rama implique tener un empaque que haya sobrepasado esta capacidad, no se va a seguir por ese camino.

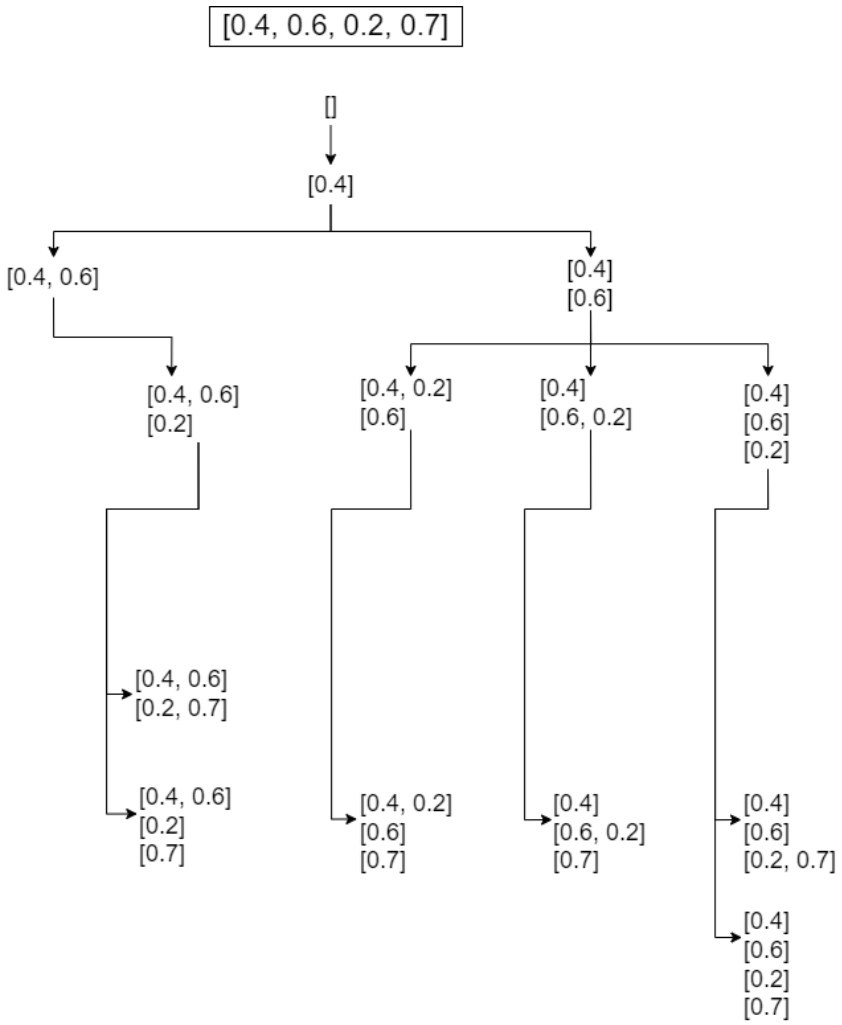

Figura 4: Algoritmo de backtracking en el que no se sigue por caminos donde hayan paquetes que superen el límite

Como podemos observar, de esta forma hay muchos caminos que, al saber que no resultarán en una respuesta válida, podemos evitar tomar.

Otra cosa que podemos tener en cuenta a la hora de recorrer nuestras opciones son los mínimos ya encontrados. Veamos la figura 4 un poco más. Si nuestro algoritmo empieza yendo hacia la izquierda, va a continuar hasta encontrar una hoja en la que hay 2 empaques, con lo que encontramos un mínimo de 2. Posteriormente, cuando el algoritmo continúe hacia la derecha y se encuentra que hay dos empaques, nos preguntamos "¿Qué es lo que puedo conseguir continuando por este camino?". Es decir, si ya encontramos una solución válida que hace uso de 2 paquetes, lo mejor que puede pasar es que en esta nueva rama que ya va 2 paquetes este número se mantenga y lleguemos a una hoja con dos paquetes, con lo que nuestro mínimo no cambiará. En el otro caso, continuar por la rama nos puede llevar, como vemos en la figura, a hojas en las que hay 3 o 4 paquetes, que no harían que el mínimo mejorara. Por ende, siguiendo con nuestra solución de **backtracking**, antes de seguir por un camino nos preguntaremos si la cantidad de paquetes que llevamos hasta ese momento es mayor o igual al mínimo, en cuyo caso regresaremos.

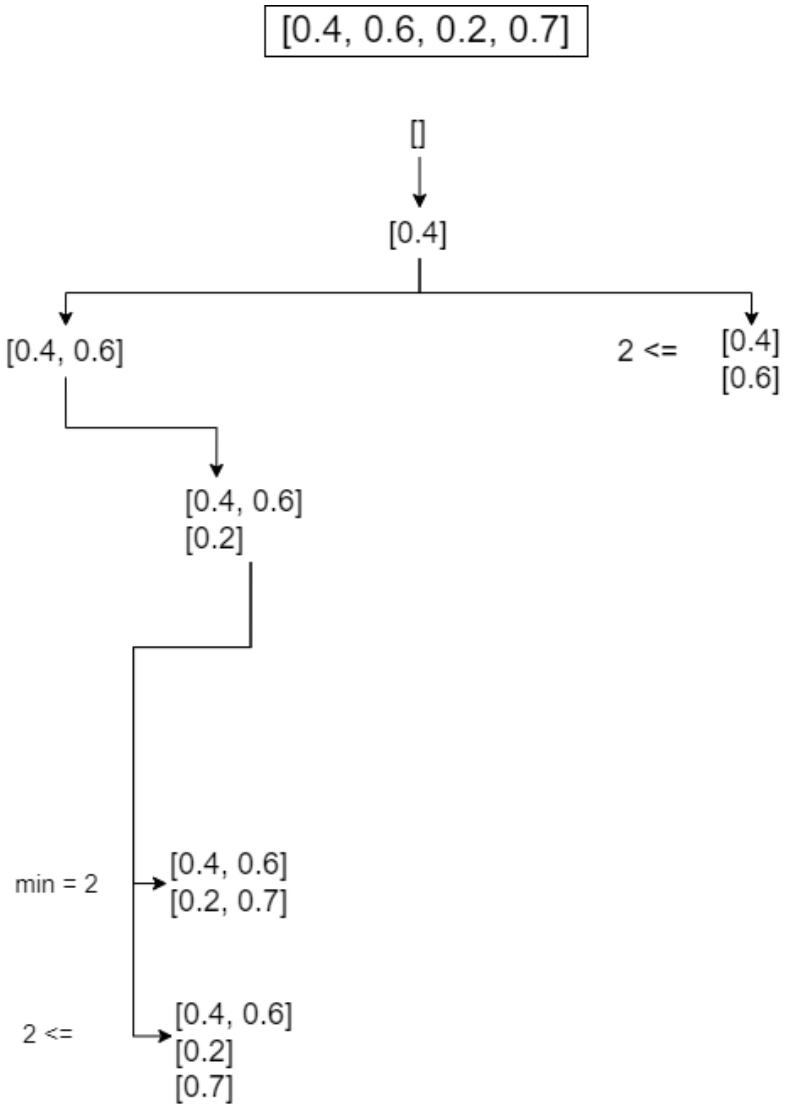

Figura 5: Algoritmo de backtracking en el que no se sigue por caminos donde hayan paquetes que superen el límite ni caminos en los que no encontraremos mejores soluciones

Podemos ver claramente todos los caminos que nos ahorramos recorrer por el hecho de tener estas condiciones que vamos chequeando al avanzar.

Cabe destacar que, a pesar de que en el caso presentado fueron muchos los caminos que nos evitamos recorrer al encontrar una solución buena al principio, esto puede no ser siempre el caso y depende del orden de los objetos. Es por eso que, en vez de inicializar el mínimo con infinito al empezar el programa y todavía no haber encontrado una solución con la cual comparar, lo que se puede hacer es inicializar el mínimo con el resultado de algún algoritmo de aproximación, el cual no tardará mucho en correr y nos puede dar rápidamente un primer límite para evitar ir por caminos muy largos al principio. Nosotros hacemos esto con uno de los algoritmos de aproximación desarrollados llamado First Fit.

También pudimos observar, al momento de hacer pruebas en el cuarto punto a desarrollar, que ordenar los objetos de mayor a menor antes de correr nuestro algoritmo de backtracking hizo que aumentara la velocidad con la que encontraba un óptimo y, por lo tanto, que el algoritmo se ejecutara más rápido.

1.2.2. Complejidad del algoritmo

Ahora que hemos explicado todas las partes importantes de nuestro programa es hora de calcular su complejidad.

Nuestro algoritmos es recursivo, sin embargo no reduce el problema a una fracción de este, con lo que no podremos utilizar el teorema maestro. Cabe destacar que, como mencionamos antes, se utiliza un algoritmo de aproximación para calcular el primer mínimo de nuestro problema, sin embargo ese algoritmo solo se llama una vez y su complejidad es despreciable comparada a la de nuestro algoritmo recursivo.

El algoritmo recursivo se llama **packaging backtracking aux**. Este recibe la lista de objetos, la posición del objeto que estamos evaluando, la cantidad mínima de paquetes para una solución encontrada hasta el momento y el estado de los paquetes en la rama actual.

Al empezar el algoritmo se hacen algunos chequeos para las condiciones de corte que son *O(1)*, luego entramos en un **for loop** que va a iterar por sobre todos los empaques disponibles en el momento, llamando de vuelta a la función recursiva. También se la llama una vez más al salir del ciclo.

Podemos denotar a la complejidad de cada llamada hacia la función como:

$$T(a, n)$$

En donde a es igual a la cantidad de empaques que hay actualmente y n es igual a la cantidad de objetos que no están asignados a ningún paquete.

Nuestro algoritmo empieza entonces en $T(0, n)$, pues al principio no hay paquetes y los objetos no están asignados.

La ecuación sería igual a:

$$T(a, n) = 0(1) + aT(a, n-1) + T(a+1, n-1)$$

Esta ecuación es así debido a que primero tendremos tantas llamadas recursivas como paquetes tengamos en el momento. Llamadas en las cuales no agregamos nuevos paquetes, pero sí decrece el número de objetos no asignados.

Por otra parte también tendremos la llamada fuera del for loop, en la cual sí se aumenta la cantidad de paquetes en uno, además de como antes hacer que la cantidad de objetos sin asignar decrezca.

Como empezamos por $T(0, n)$ entonces nuestra ecuación inicial, es:

$$T(0, n) = 0(1) + 0 * T(0, n-1) + T(1, n-1)$$
$$T(0, n) = 0(1) + T(1, n-1) = 0(1) + 1 * T(1, n-2) + T(2, n-2)$$

Vamos avanzando así, cambiando las ecuaciones que surgen por sus equivalentes intentando llegar al caso base

$$T(1, n-2) = 0(1) + 1 * T(1, n-3) + T(2, n-3)$$
$$T(2, n-2) = 0(1) + 2 * T(2, n-3) + T(3, n-3)$$

Reemplazando:

$$T(0, n) = 0(1) + 1 * (0(1) + 1 * T(1, n-3) + T(2, n-3)) + 0(1) + 2 * T(2, n-3) + T(3, n-3)$$

$$T(0, n) = 0(1) + 2 * O(1) + 1^2 * T(1, n-3) + 1 * T(2, n-3)) + 2^1 * T(2, n-3) + T(3, n-3)$$

Vemos que en el primer llamado a nuestra función teníamos complejidad O(1) y luego hacíamos más llamados recursivos. Pero como en cada llamado recursivo se hace lo mismo, entonces por cada llamado recursivo vamos a tener también una complejidad O(1). Podríamos ver esto de la siguiente manera, al hacer un for loop que tiene en su interior operaciones O(1), si se va a hacer **n** veces entonces decimos que tenemos complejidad O(n). En nuestro caso, cuando empieza el primer loop de todos, se tienen operaciones O(1) y llamados recursivos a otras funciones que también tendrán operaciones O(1). Con lo cual podemos ver que la cantidad de veces que haremos estas operaciones es igual a la cantidad de llamados recursivos que tengamos en total.

Como estamos planteando el peor escenario posible, este será el escenario en el que no hay ninguna poda, que se da cuando la menor cantidad de empaques que podemos tener es igual a la cantidad de objetos. Este caso se da cuando cada objeto tiene un valor mayor a 0.5.

Avancemos más en los llamados recursivos:

$$T(1, n-3) = O(1) + 1 * T(1, n-4) + T(2, n-4)$$
$$T(2, n-3) = 0(1) + 2 * T(2, n-4) + T(3, n-4)$$
$$T(3, n-3) = 0(1) + 3 * T(3, n-4) + T(4, n-4)$$

Reemplazamos nuevamente

$$\begin{aligned} T(0,n) = {} & 0(1) + 2 * O(1) + 1^2 * (0(1) + 1 * T(1, n-4) + T(2, n-4)) + 1 * (0(1) \\ & + 2 * T(2, n-4) + T(3, n-4)) + 2^1 * (0(1) + 2 * T(2, n-4) + T(3, n-4)) \\ & + O(1) + 3 * T(3, n-4) + T(4, n-4) \end{aligned}$$

$$\begin{aligned} T(0,n) = {} & 0(1) + 2 * O(1) + 5 * O(1) + 1^3 T(1, n-4) + 1^2 * T(2, n-4) + 1 * 2 * T(2, n-4) \\ & + 1 * T(3, n-4) + 2^2 * T(2, n-4) + 2 * T(3, n-4)) + 3 * T(3, n-4) + T(4, n-4) \end{aligned}$$

Si sumamos los T que son iguales

$$T(0,n) = 0(1) + 2 * O(1) + 5 * O(1) + T(1, n-4) + 7 * T(2, n-4) + 6 * T(3, n-4) + T(4, n-4)$$

Esto va a seguir hasta que la cantidad de "niveles"que hayamos bajado en el árbol sea igual a la cantidad de elementos que teníamos en un principio, por lo que el último elemento en la última fórmula sería $T(n, n-n)$.

Si seguimos avanzando se van a sumar cada vez números más grandes de llamados recursivos. Lo que vimos en estas ecuaciones se puede ver claramente en la *figura 3*. Vemos que si tuviéramos un único elemento haríamos solo un llamado, al tener dos hacemos dos llamados recursivos extra, con 3 elementos ya hacemos 5 más y con 4 elementos hacemos 15 más. Si tuviéramos 6 elementos haríamos 52 llamados más y si tuviéramos 7 elementos haríamos 203 más.

Analizando esta secuencia de números nos podemos dar cuenta de que son los **números de Bell**. El n-éismo número de Bell representa la cantidad de formas distintas en las que se pueden dividir los elementos de un conjunto con n elementos. Tiene todo el sentido del mundo que la cantidad de operaciones que hagamos aumente en ese rango, pues en cada nivel del árbol tenemos la cantidad de formas de dividir un conjunto con la cantidad de elementos igual al nivel del árbol.

Vemos entonces que nuestra complejidad será igual a la sumatoria de los primeros **n** números de Bell, siendo **n** la cantidad de elementos que tengamos que poner en paquetes. El n-ésimo número de Bell se calcula de la siguiente manera, siendo $B_0 = B_1 = 1$:

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_n$$

Nuestra complejidad sería entonces

$$T(0, n) = O(\sum_{k=1}^{n} B_k)$$

Esta complejidad no llega a ser factorial, pues podemos ver que cuando tenemos 5 elementos hacemos $1 + 2 + 5 + 15 + 52 = 75$, mientras que 5! es 120. Y estos números no hacen más que distanciarse a medida aumenta la cantidad de objetos. Sin embargo, es muy claro que nuestro algoritmo tiene una **complejidad exponencial** y que no hacen falta muchos elementos para que el algoritmo tenga un tiempo de ejecución muy grande. Si necesitáramos una cota superior para nuestra complejidad con tal de no incluir las fórmulas con los números de Bell, la más apropiada sería $O(n!)$, siendo n la cantidad de elementos, pues sabemos que nunca vamos a superar esta complejidad.

Ahora hagamos mediciones en función del tiempo

1.2.3. Tiempo de ejecución del algoritmo en función de la cantidad de objetos

En un primer momento probamos sin ordenar los objetos antes de correr el algoritmo. Dado que el tiempo de ejecución de nuestro algoritmo aumenta de manera exponencial, no pudimos probar con cantidades grandes de datos. Lo que hicimos fue crear 3 sets de datos por cada cantidad de objetos, desde 1 objetos hasta 23 objetos. Hacemos que nuestro algoritmo resuelva los 3 casos y luego promediamos el tiempo que tardó para tener una mejor representación de la realidad. Al tener una cantidad mayor de datos el tiempo de ejecución de nuestro algoritmo aumentaba muchísimo, haciendo que no fuera viable que resolviera esos problemas 3 veces.

El tiempo que demoró en obtener una respuesta por cada cantidad de objetos que se probó fue:

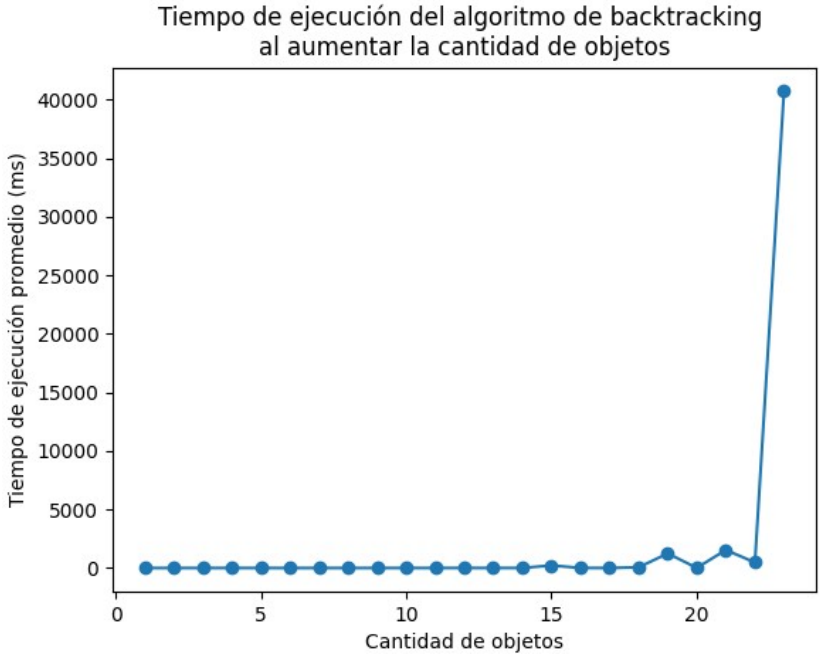

Figura 6: Medición del tiempo promedio de ejecución del algoritmo de backtracking hasta 23 objetos

En el caso de nuestro algoritmo, para las primeras cantidades lo logró hacer muy rápido, probablemente porque nuestra primera aproximación de la cantidad mínima de empaques dio un buen resultado y se hizo una poda de casi todas las ramas.

Mientras más elementos vamos agregando, menos probable es encontrar un mínimo que haga una poda así de considerable al empezar el problema. Vemos que a partir de 19 elementos empiezan a haber picos, probablemente porque uno de los 3 casos se tardó al seguir encontrando posibles soluciones y eso hizo aumentar el promedio.

Finalmente con 23 elementos hay un pico muy grande y si seguimos aumentando la cantidad de elementos el tiempo que tarda se vuelve muy grande como para hacer múltiples mediciones.

Podemos ver que los resultados concuerdan con lo analizado en la sección de complejidad,pues el tiempo de ejecución aumenta mucho mientras más objetos tenemos, aunque con las podas esto pueda no notarse para pocos objetos.

Como se mencionó anteriormente, después de hacer el punto 4 nos dimos cuenta que los tiempos mejoraban notablemente al ordenar los elementos de forma decreciente antes de correr nuestro algoritmo. Al probar nuevamente con eso, pudimos llegar mucho más lejos, aunque ahora solo hicimos una corrida para cada cantidad de objetos, siendo los tiempos de ejecución los siguientes comparados al algoritmo first fit:

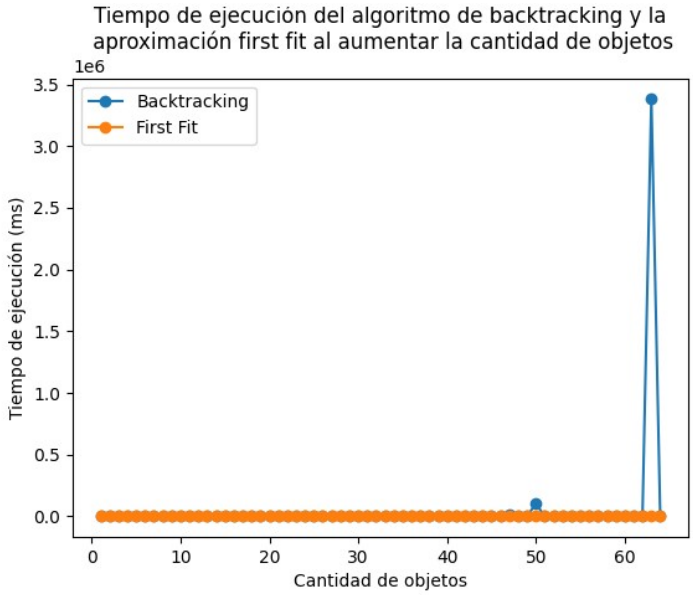

Figura 7: Medición del tiempo promedio de ejecución del algoritmo de backtracking hasta 64 objetos

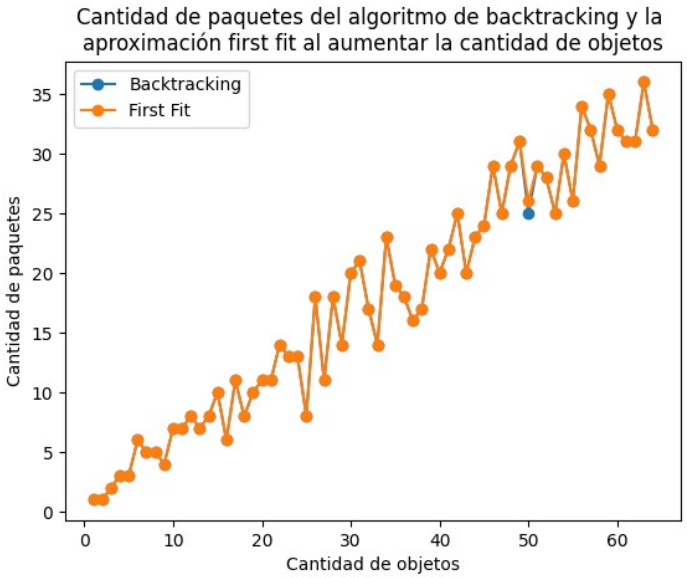

Figura 8: Comparación de la cantidad de paquetes devuelta por el algoritmo first fit y el algoritmo de backtracking

Podemos ver que al correr nuestro algoritmo después de ordenar los elementos este es capaz de ejecutarse más rápido y llegar más lejos, pudiendo hacer hasta 64 objetos, parando aquí porque al avanzar más el algoritmo tardaba horas sin dar un resultado para una sola corrida.

También vemos que el algoritmo First Fit, en la gran mayoría de casos, nos está dando el óptimo, con lo cual en muchos casos nuestro algoritmo de backtracking desde el principio puede hacer una poda imporatante, haciendo que vaya tan profundo en el árbol de posibles soluciones y evitando mucho tiempo de ejecución. Esto nos dice unas cuantas cosas.

La primera es cuanto afecta encontrar un mínimo que sea bueno temprano en nuestro algoritmo. Al ordenar los elementos es más probable que demos antes con este mínimo y luego será más fácil descartar candidatos, ahorrándonos mucho tiempo en la corrida del programa.

Otro punto a destacar es que ver un patrón de crecimiento específico en los gráficos resulta complicado, pues hay veces en los que va muy rápido al encontrar una buena solución temprano, pero en otros se tarda muchísimo y esa cantidad de tiempo no hace más que aumentar a medida que avanzamos, muchas veces saltando de rápido a muy lento con solo aumentar un elemento. Por lo que en definitiva es un algoritmo cuyo tiempo de ejecución varía mucho dependiendo de los objetos presentes. Recordemos que en los primeros gráficos estamos haciendo 3 versiones de cada cantidad de elementos y luego promediándolas, por lo que incluso los picos son una de las 3 versiones que corrió muy lento y las otras dos corriendo rápido, pero el promedio da un punto medio.

1.3. Algoritmo de aproximación

En este punto se implementó el algoritmo para aproximar la solución del problema de empaquetamiento propuesto de la manera descrita en la consigna.

La manera en la que se implementó fue haciendo un doble ciclo while. El primero sirve para añadir un nuevo paquete a nuestra lista de paquetes, mientras que el segundo sirve para ir poniendo el siguiente objeto en la lista en el último paquete, entonces corta cuando ya no hay espacio, lo que hace que volvamos al primer while y se agregue un nuevo paquete al cual meterle objetos.

La complejidad de este algoritmo, el cual recibe una lista de elementos de tamaño **n**, se puede calcular viendo que, a pesar de que hacemos dos ciclos while, ambos tienen como condición que se llegue a un tope de la misma variable que representa una iteración sobre la lista de elementos. Por lo que se van a realizar **n** acciones de complejidad igual a lo que se haga dentro del segundo ciclo while, o esa misma complejidad sumada a lo que se hace en el primer cilo while.

En el segundo ciclo while lo que se hace agregar un elemento al final de una lista, lo cual tiene complejidad **O(1)**. En el primer ciclo while lo que se hace también es agregar un elemento a una lista, por lo que también es **O(1)**.

Esto nos dice entonces que la complejidad de nuestro algoritmo es **O(n)**.

Como podemos observar, la complejidad de esta aproximación es muchísimo menor a la del algoritmo que consigue la solución óptima por backtracking. A continuación vamos a darle los mismos sets de datos a ambos algoritmos para comparar el tiempo que se demoran en correr y la diferencia de empaques que cada uno asigna.

Probamos con sets de datos que tenían diferentes cantidades de objetos. Por cada cantidad se hicieron 3 versiones de los datos para promediar el resultado y compararlo mejor.

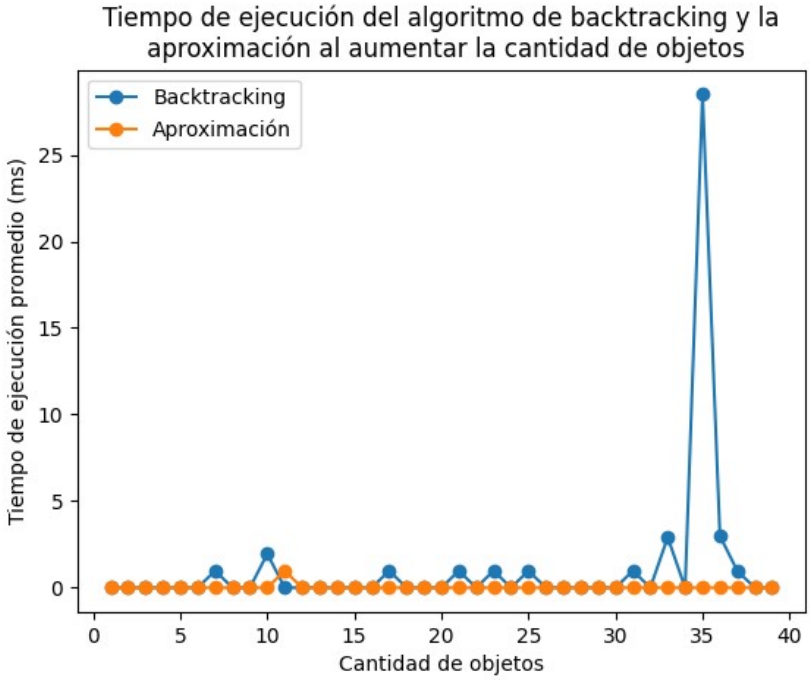

Figura 9: Comparación del tiempo promedio que demora el algoritmo de backtracking con el algoritmo de aproximación por la cantidad de objetos

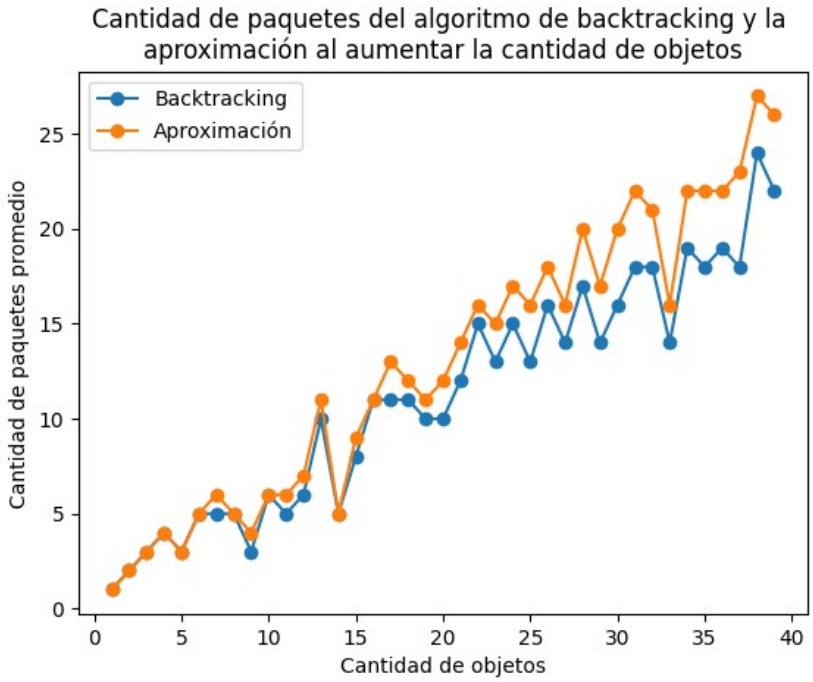

Figura 10: Comparación de la cantidad de paquetes que nos da el algoritmo de backtracking con el algoritmo de aproximación por la cantidad de objetos

Podemos apreciar que en los primeros casos la cantidad de paquetes mínimos propuesta por cada algoritmo no difiere mucho. Sin embargo, llega un punto en el que el algoritmo de backtracking siempre encuentra que los objetos se pueden poner en una cantidad de empaques menor a la que el algoritmo de aproximación nos dijo.

En cuanto a tiempo de ejecución, con cantidades pequeñas de objetos el algoritmo de backtracking pudo ejecutarse en un tiempo muy corto. Esto puede deberse a que se llega a una buena solución pronto y gracias a la poda de caminos el algoritmo se termina de ejecutar más rápido. Sin embargo cuando no encuentra esta solución temprano el tiempo que se puede dilatar bastante como se vió en el punto anterior.

Ahora, siguiendo con la consigna, tenemos que calcular la cota $r(A)$ tal que, siendo $A(I)$ la solución aproximada del problema y $z(I)$ la solución óptima, $\frac{A(I)}{z(I)} \leq r(A)$.

Para calcular esta cota, habrá que pensar en el peor caso posible. Pero antes, pensemos en qué sucede si nuestro algoritmo óptimo tiene dos empaques llenos, en caso de dividir estos empaques y reordenar sus elementos, en cuanto podríamos hacer aumentar la cantidad de paquetes usando nuestra aproximación. Digamos que en el primera paquete tenemos 0.6, 0.2 y 0.1, y en el otro paquete tenemos 0.4 y 0.6. Primero tenemos que pensar que cada empaque puede tener a lo sumo un elemento que es mayor a la mitad de la capacidad. Digamos que nuestra lista está en el siguiente orden [0.6, 0.6, 0.4, 0.2, 0.1]. Si usáramos nuestra aproximación tendríamos 3 paquetes, lo cual es esperable, pues si todos estos elementos cabían en dos paquetes, si ponemos en un paquete un objeto que ocupa más de la mitad del espacio y en otro otro de estos, los objetos que sobran podrán sumar como mucho la capacidad máxima de un paquete.

Ahora pensemos en el caso en el que tenemos 3 paquetes que almacenan los objetos [0.5, 0.5, 0.5, 0.5, 0.1, 0.1, 0.1, 0.1, 0.6]. En nuestro algoritmo óptimo pondríamos los 4 0.5 en dos paquetes y el resto de objetos en otro paquete. Sin embargo, para nuestro algoritmo de aproximación el peor escenario sería que se lo diéramos en este orden [0.5, 0.1, 0.5, 0.1, 0.5, 0.1, 0.5, 0.1, 0.6], pues en ese caso terminaría con 5 paquetes. Podemos ver que para este algoritmo, siempre al intentar encontrar el peor caso nos podemos apoyar en este, en el que un pequeño número se le agrega a un objeto que ocupa la mitad del empaque y el siguiente objeto que también ocupa exactamente la mitad ya no puede entrar. Esto resulta en que si con el algoritmo óptimo teníamos una cantidad N de paquetes, el algoritmo de aproximación en su peor caso tendrá $2N - 1$ paquetes. Por lo tanto $r(A)$ es igual a $2N - 1$.

1.4. Algoritmos de aproximación adicionales

Vimos en el punto anterior un algoritmo de aproximación que, al investigar su forma, vimos que era un algoritmo del tipo Next Fit, en el que solo veíamos el último paquete abierto. Para crear un algoritmo diferente lo primero que se nos vino a la cabeza fue pensar en hacer uno que, a diferencia del algoritmo Next Fit, no ignorara todo el espacio libre que quedaba de los paquetes que se cerraron prematuramente. Por lo que se hizo un algoritmo que en vez de fijarse en el último paquete, se fije en todos los que hay en un determinado momento y vea cual es el primero en el que lo puede poner. Al investigar más vimos que a este tipo de algoritmos se los conoce como First Fit.

En este algoritmo, por cada elemento que tengamos vamos a recorrer todos los empaques disponibles en el momento. La peor cantidad de paquetes que podemos tener es igual a la cantidad total de objetos, por lo que si estamos en el objeto n, lo peor que puede pasar es que tengamos que buscar entre n-1 paquetes. Siendo la complejidad una sumatoria de la forma $\sum_{k=1}^{n} n*(n-1)$. Esta sumatoria es conocida y su resultado es $n*(n+1)$. Con lo que podemos ver que la complejidad del algoritmo First Fit será **cuadrática**, a diferencia de Next Fit que es **lineal**.

Pero también vimos que, poner los elementos en el primer empaque en el que entren puede que tampoco sea la mejor aproximación que podemos hacer. Quisimos también tener una aproximación que tenga un criterio sobre qué empaque de entre los que tienen suficiente espacio para almacenar a nuestro objeto elegir. Una de las aproximaciones de este tipo que pudimos ver es Best Fit, que intenta poner el objeto en el empaque que sea el que, sumándole el objeto actual, se acerque más al límite sin superarlo.

Para este algoritmo también lo que se hizo fue ordenarlo de forma descendiente antes de correrlo pues teóricamente sacarse los elementos más grandes de encima al principio lo debería hacer funcionar mejor, pues tendremos más empaques muy llenos con pocos elementos y otros paquetes con muchos elementos pequeños.

Ordenar el vector de elementos es *O(nlog(n))*, sin embargo esta complejidad no se toma en cuenta al ser mucho menor que la del algoritmo en sí. Este algoritmo en teoría tiene la misma complejidad que First Fit, **cuadrática**, sin embargo First Fit va a parar de iterar los paquetes cuando encuentra uno en el que el objeto quepa, mientras que Best Fit los va a iterar todos siempre, con lo que aunque el peor caso de ambos tiene la misma complejidad, es esperable que Best Fit tarde más en promedio.

Aprovechando el hecho de que estás son aproximaciones con complejidades manejables, podemos hacer pruebas con muchos elementos.

Primero se hizo una prueba con 40 elementos para ver inicialmente como se comparaban los resultados.

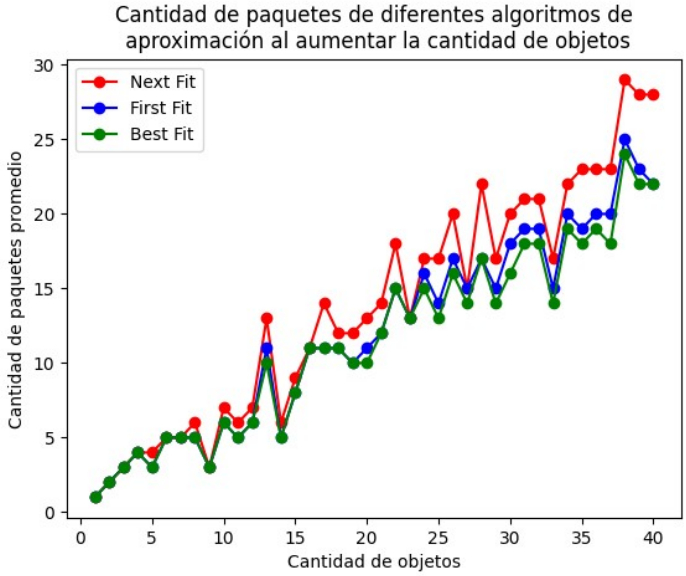

Figura 11: Comparación de cantidad de paquetes asignados por diferentes algoritmos de aproximación

Podemos apreciar como en los primeros casos dan resultados iguales, pero luego el algoritmo Next

Fit empieza a dar constantemente peor y los algoritmos First Fit y Best Fit empiezan a dar resultados similares, siendo Best Fit el que da los mejores.

Si ahora probamos con 2000 elementos obtenemos los siguientes resultados.

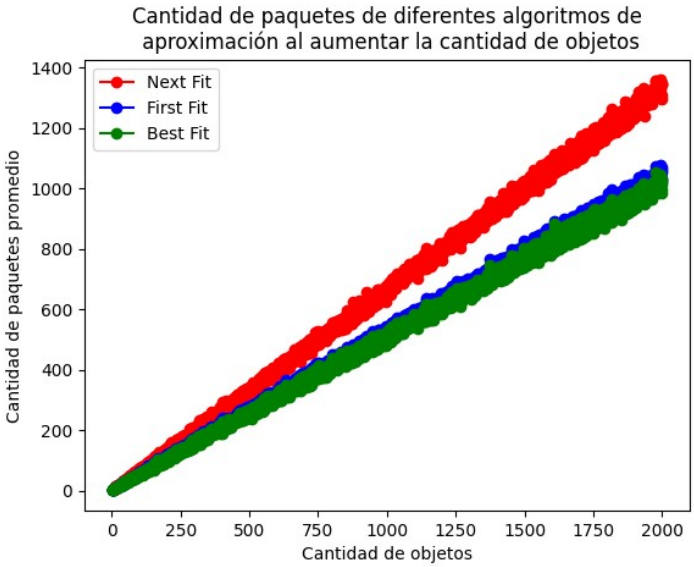

Figura 12: Comparación de cantidad de paquetes asignados por diferentes algoritmos de aproximación

Vemos que la relación de los resultados se mantiene, siendo los resultados del algoritmo de Next Fit notablemente mayores que los otros dos algoritmos. Vemos que los algoritmos de First Fit y Best Fit tienen resultados muy parecidos, sin embargo Best Fit pareciera ser constantemente algo mejor.

Si queremos ver mejor la relación entre los resultados con números altos podemos verificarlo en la siguiente imagen

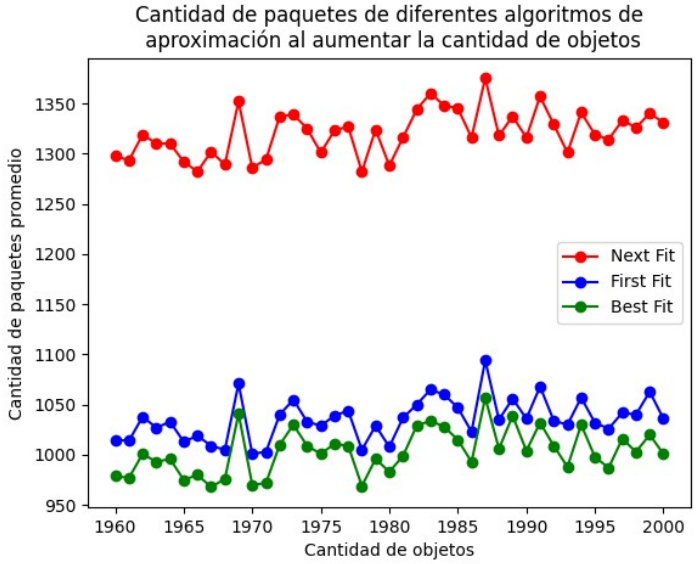

Figura 13: Comparación de cantidad de paquetes asignados por diferentes algoritmos de aproximación

Los tiempos de ejecución de los algoritmos son los siguientes:

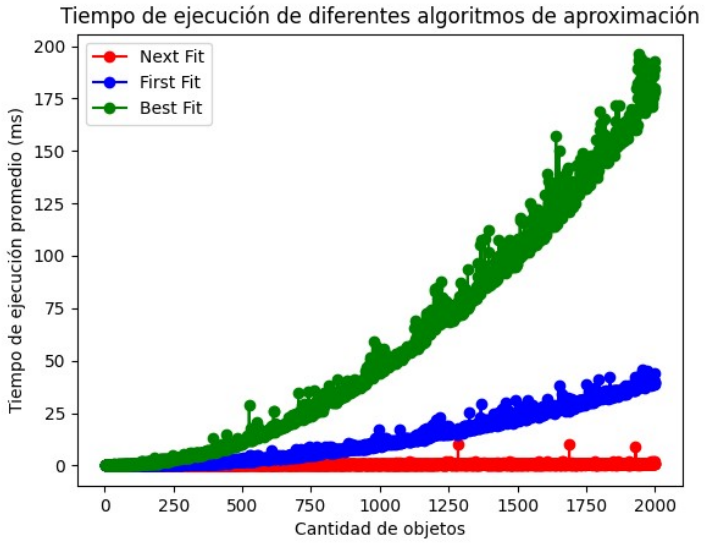

Figura 14: Comparación de los tiempos de ejecución de los algoritmos de aproximación

Podemos apreciar que, acordé a lo que anticipamos previamente, la complejidad del algoritmo First Fit es lineal, la del algoritmo First Fit es cuadrática en su peor caso pero en la realidad es mejor y el Best Fit es el más costoso siendo claramente cuadrático.

Sin embargo, todavía hay una cosa que podríamos probar. Actualmente vemos que el algoritmo que nos dio mejores resultados es el de Best Fit, sin embargo, podríamos probar qué pasaría si al igual que en el caso de Best Fit, ordenáramos los elementos antes de correr el algoritmo para Next Fit y First Fit.

Al hacerlo obtenemos los siguientes resultados.

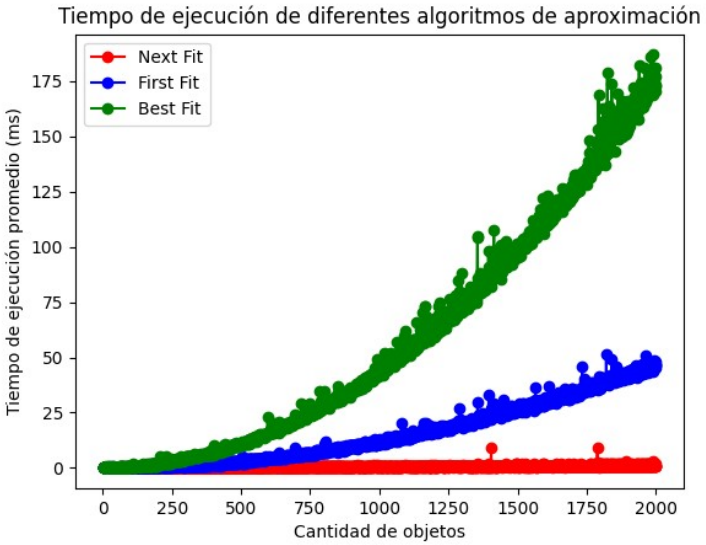

Figura 15: Comparación de los tiempos de ejecución de los algoritmos de aproximación con objetos ordenados decrecientemente

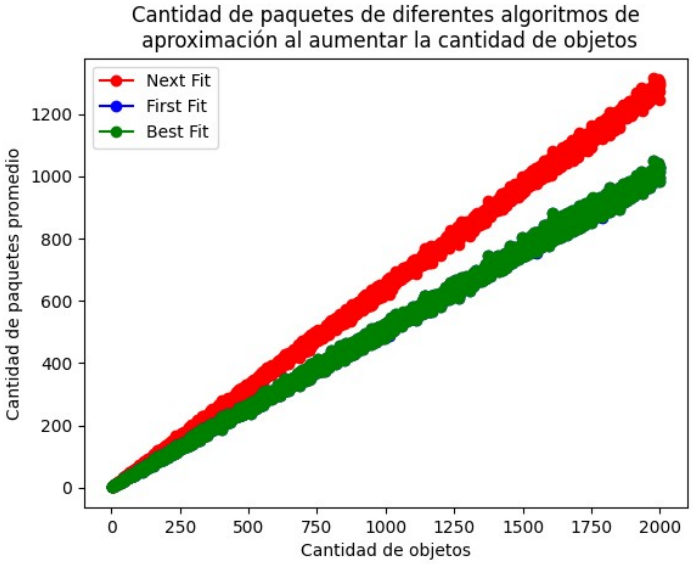

Figura 16: Comparación de cantidad de paquetes asignados por diferentes algoritmos de aproximación con objetos ordenados decrecientemente

Vemos que los tiempos de ejecución no varían, sin embargo, ahora la cantidad de objetos que nos devuelven el algoritmo First Fit y el Best Fit son las mismas en la mayoría de los casos. Hagamos un acercamiento para analizarlo mejor.

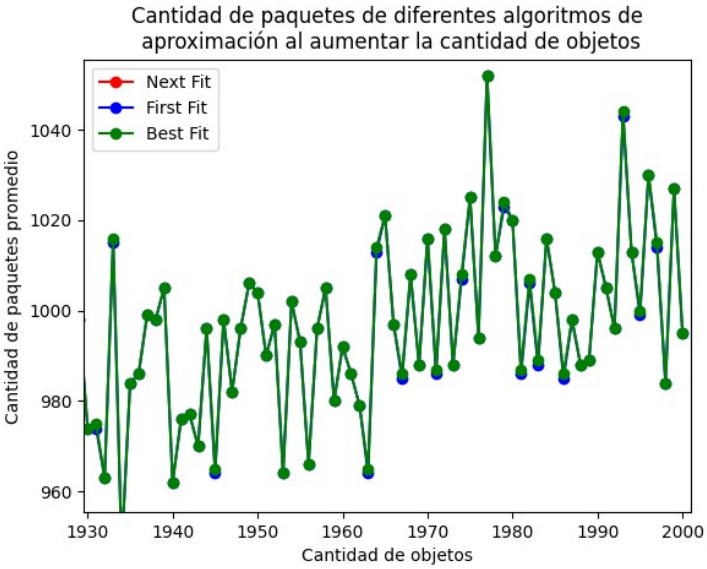

Figura 17: Comparación de cantidad de paquetes asignados por diferentes algoritmos de aproximación

Como podemos apreciar, en la mayoría de los casos Best Fit y First Fit nos dan el mismo resultado, sin embargo, hay veces en las que First Fit nos da mejores resultados. Teniendo en cuenta los mejores resultados y el hecho de que el tiempo de ejecución del algoritmo es notablemente inferior, podemos concluir que de todos los algoritmos probados, el que nos da mejores resultados es el de First Fit con los objetos ordenados de manera decreciente.