Universidad de Buenos Aires - FIUBA
75.29 Teoría de Algoritmos 1
TP2

Pernin Alejandro, *Padrón Nro. 92.216*
apernin@fi.uba.ar

Monsech Santiago, *Padrón Nro. 92.968*
smonsech@fi.uba.ar

23 de Mayo de 2023

Índice

1. Demostración NP-Complete

Para mostrar que el problema del empaquetamiento es NP-Completo, debemos validar los puntos:

- El problema debe estar en NP
- El problema debe poder ser transformado en un problema de decisión y debe poder ser reducido por otro que es NP-Completo

Para el primer punto, podemos ver que dado una cantidad de productos y un cantidad de paquetes, podemos validar la solución obtenida en tiempo polimonial, por lo que esto nos indicar que el problema vive en el espacio NP.

Para el segundo punto, debemos partir de un problema que sabemos que sea NP-Completo, que pueda ser reducido al problema del empaquetamiento y si podemos encontrar si existe la solucion del problema NP-Completo reducido con el algoritmo de empaquetamiento, entonces podemos decir que el problema del empaquetamiento es al menos tan difícil que el problema NP-Completo, entonces su dificultad queda acotada por un problema NP-Completo, podremos probar que el problema del empaquetamiento es NP-Completo.

Para nuestro análisis utilizamos el ***Problema de Partición***, este es un problema **NP-Completo**, el problema consiste en si se puede a partir un multiconjunto S de números enteros y obtener una partición S_1 y S_2 de S de tal manera que la suma de los elementos de cada subconjunto sea la misma,

$$\sum_{x \in S} \frac{x}{2} = \sum_{x \in S_1} x = \sum_{x \in S_2} x$$

.

Decidimos elegir este problema por que ambos tienen esta idea de particionar elementos en diferentes subconjuntos tal que tienen que cumplir alguna condición. En el problema del empaquetamiento partimos de un multi conjunto de n números $S = \{a_1, a_2, ..., a_n\}$, con paquetes de capacidad C y un numero de k paquetes.

Para poder hacer la prueba partimos de una instancia del problema de partición, tal que teniendo un multi conjunto de números $S = \{a_1, a_2, ..., a_n\}$ que mapeamos uno a uno cada numero con un producto a empaquetar, $C = 1$ y $k = 2$ por lo que se pueden normalizar los números enteros con $n_i = \frac{2 * c_i}{\sum_{j=1}^{n} c_j}$, tal que $C = \frac{(\sum_{\forall n_i} n_i)}{2}$.

Entonces si existe una solución donde los productos se pueden empaquetar en k=2 paquetes de exactamente capacidad C=1, entonces existe una solución para el problema de Partición. Por ende queda demostrado que el problema del Empaquetamiento es al menos tan difícil como el de Partición y como este ultimo es NP-Completo, entonces el primero sera NP-Completo.

Una cuestión mas a mencionar es que el algoritmo para hallar la solución del problema del empaquetamiento es una aproximación, en la siguiente sub sección mostramos que la relación de la aproximación $r(A)$ es mayor a 1, quiere decir que si existe una solución al problema de la partición, a lo sumo el problema del empaquetamiento empaquetara en a los sumo 3 paquetes en vez de en 2.

2. Algoritmo de Aproximación

Las futuras secciones se basarán en el algoritmo de aproximación, por lo cuál resulta prudente hacer un análisis preliminar del mismo. Considerando un conjunto de N elementos
se observa que la complejidad temporal del algoritmo es de $O(N)$

Algorithm 1 Aproximación Empaquetado

```
1: Abrir un paquete
2: while Hayan elementos sin empaquetar do                          ▷ O(N)
3:     if El objeto entra en el envase then
4:         Se agrega el objeto al envase y se continua con el siguiente objeto   ▷ O(1)
5:     else
6:         Se cierra el envase actual y se abre uno nuevo
7:         Se agrega el objeto y se continua con el siguiente       ▷ O(1)
8:     end if
9: end while
10: Devolver los paquetes
```

3. Fuerza Bruta

Para el algoritmo de fuerza bruta emplearemos el método de empaquetamiento por aproximación dado en el enunciado, notando que para un mismo conjunto de elementos el orden de los mismos puede afectar el resultado. Podemos considerar que para este algoritmo de aproximación, para un conjunto cualquiera de N elementos existe un orden específico de los mismos para el cuál el algoritmo nos brinda una solución óptima.

Por lo cuál nuestro algoritmo de fuerza bruta consiste en obtener todas las permutaciones posibles de N elementos y con estas permutaciones alimentar el algoritmo de aproximación y guardar el menor de los resultados (aquel que emplee la menor cantidad de paquetes).

Sabiendo que para N elementos la complejidad del a aproximación es $O(N)$, resta analizar cuantas permutaciones posibles existen para un conjunto de N elementos. Es conocido que este valor es $N!$, realizaremos la demostración:

Para un conjunto de N elementos, una permutación se armará de la siguiente manera

- **Primer Elemento (índice 0)**: Se elije un elemento de entre el conjunto de N
- **Segundo Elementos (índice 1)**: Se elije un elemento de entre los N-1 restantes
- **Tercer Elemento (índice 2)**: Se elije de los N-2 restantes
- **Elemento i (índice N-(i-1))**: Se elije de entre los N-i restantes
- **Elemento N (índice N-(N-1))**: Se elije el único elemento restante

es evidente entonces que la cantidad de variaciones es $N! = N \cdot (N-1) \cdot (N-2) \cdots 1$

Combinando ambos análisis podemos concluir que nuestro algoritmo por fuerza bruta tendrá una complejidad de $O(N \cdot N!)$, siendo que para N grandes $N! \gg N$ el factor dominante será el factorial, por ende $O(N!)$

3.1. Pruebas de Ejecución

Para la realización de las pruebas de ejecución se emplearon múltiples llamadas variando la cantidad de elementos, asimismo para una misma cantidad de elementos se realizaron diversas llamadas con conjuntos de datos distintos para obtener cierta variabilidad, promediando luego los resultados.

Dado que el tiempo de ejecución crece factorialmente con el incremento de la cantidad de elemento, se tomaron las siguientes consideraciones

- Empleamos una cantidad acotada de Ns
- Para un mismo N realizamos sólo 3 variaciones del conjunto de datos
- Se empleó un Pool de procesos de Python para agilizar la ejecución de las pruebas

3.1.1. Resultados

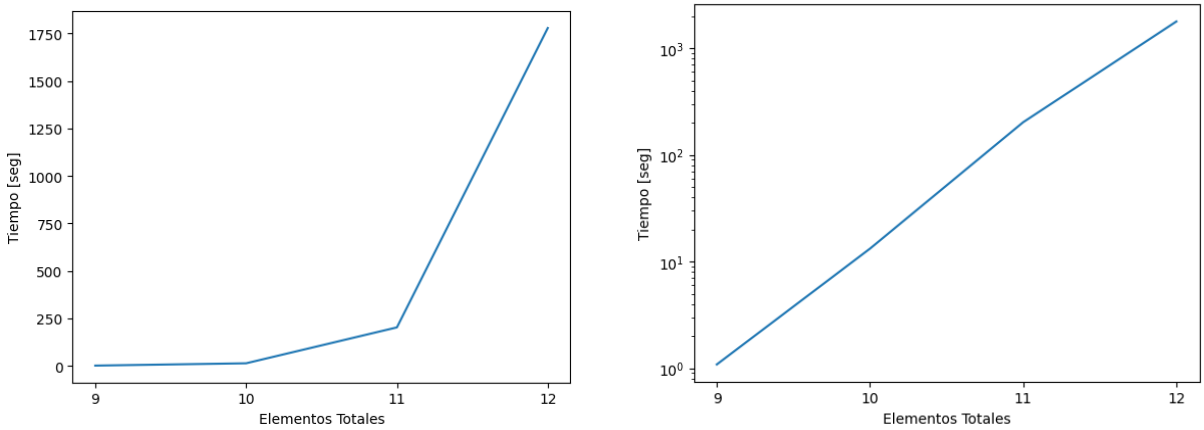
Se observa que los resultados son consistentes con la complejidad obtenida en la sección anterior.

n	times (seg)
9	1.216916
9	1.137836
9	0.898380
10	7.831767
10	15.780019
10	16.030042
11	210.776504
11	215.460243
11	180.665471
12	1776.591041
12	1750.481009
12	1810.784472

Cuadro 1: Ejecuciones de Prueba y sus tiempos de ejecución

n	avg
9	1.084377
10	13.213943
11	202.300740
12	1779.285507

Cuadro 2: Promedios de ejecución



(a) Escala Lineal

(b) Escala Logaritmica

Figura 1: Ejecuciones de Fuerza Bruta

4. Análisis Aproximación

4.1. Complejidad

La complejidad del algoritmo de aproximación ya fue analizada en la seccion 2

4.2. Cota de Aproximación

Siendo I una instancia cualquiera del problema de empaquetamiento, $A(I)$ la solución aproximada, $z(I)$ la solución óptima, definiremos $r(A) \leq \frac{A(I)}{z(I)}$ como una cota de la relación entre la aproximación y el

óptimo de todas las instancias posibles. En esta sección intentaremos encontrar un valor a dicha cota.

Comenzaremos buscando una instancia I para el cuál $A(I)$ resulte la peor respuesta posible. Conociendo como opera la aproximación, contemplemos el siguiente escenario de un conjunto T de N elementos

$$T = \{T_1, T_2, T_3, ..., T_n\}$$

consideremos ahora los siguientes elementos alternados

$$T = \{T_1 = 1 - \epsilon, T_2 = 2\epsilon, T_3 = 1 - \epsilon, T_4 = 2\epsilon, ...\} \; \forall \epsilon > 0$$

podemos entonces agrupar los elementos entre los de índice par e impar

$$T_{2k-1} = 1 - \epsilon \wedge T_{2k} = 2 \cdot \epsilon \;\; , k \in [1, N/2]$$

resulta evidente que la aproximación se comportará de la siguiente manera

- Se agrega T_1 al envase
- T_2 no entra en el envase, por lo cuál se cierra el actual y abre uno nuevo, colocando T_2 en él
- T_3 no entra en el envase, se cierra el actual, abre uno nuevo y coloca T_3 en él
- Continúa de la misma forma hasta concluir todos los objetos

Dado este conjunto de datos determinado, la aproximación nos dará como resultado que cada objeto se empaquetará en un envase exclusivo. Por ende $A(I_N) = N$

Resta ahora para este mismo escenario, obtener el resultado óptimo $z(I_N)$, para ello vamos a proponer como hipótesis que ϵ es los suficientemente pequeño como para que todos los $T_{2k} = 2 \cdot \epsilon$ entran en un mismo envase, es decir

$$\sum_{k=1}^{N/2} 2 \cdot \epsilon \leq 1$$

a su vez resulta evidente que para envasar los restastes elementos $T_{2k-1} = 1 - \epsilon$ se requerirán $N/2$ envases (tantos envases como T_{2k-1} hayan).

Finlamente entonces obtenemos que $z(I_N) = N/2 + 1$.

$$r(I_N) = \frac{A(I_N)}{z(I_N)} = \frac{N}{N/2 + 1} < 2$$

concluimos entonces que la cota superior está dada por

$$r(I_N) = \frac{A(I_N)}{z(I_N)} < 2$$

4.3. Demostración

Asumamos existe una instancia K tal que $r(K) > r(I_N)$, en particular analicemos $r(K) = 2$. Dicho valor significaría que $A(K) = 2 \cdot z(K)$.

Siendo z_i y A_i los envases tamaños de la solución exacta y aproximada respectivamente, resulta evidente que la relación implica que cada envase A_i se encuentra a mitad de capacidad en comparación a z_i, siendo que la capacidad máxima de ambos es 1, implicaría que los envases A_i se encuentran a mitad de capacidad. Esto significa que habiendo un envase a mitad de capacidad y teniendo un objeto de tamaño $W < 1/2$ el algoritmo decidió abrir un nuevo envase en vez de colocarlo en el envase existente; lo cuál es un absurdo.

4.3.1. Gráficos

Se realizaron múltiples pruebas de cantidad variable y aleatoria de elementos, guardando la relación entre la solución aproximada y la exacta, logrando los siguientes resultados

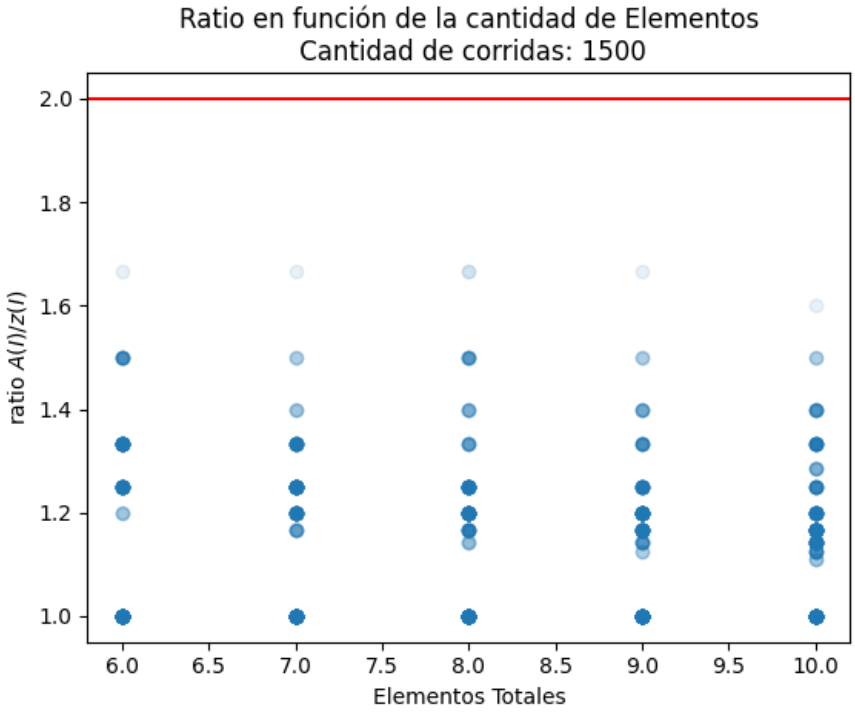


Figura 2: Análisis Algoritmo

group	count	perc
$r = 1$	684	0.456000
$1 < r \leq 1{,}5$	810	0.540000
$1{,}5 < r \leq 2$	6	0.004000

Cuadro 3: Agrupamiento según ratio

NOTA: Se limitó el tamaño de los conjuntos a 10 para evitar corridas largas, recordando lo expuesto en el cuadro 2 donde una corrida de 11 elementos tarda aproximadamente 3 minutos.

Como es lógico el algoritmo de aproximación nunca obtiene un resultado mejor que el óptimo, sin embargo en el 45 % de los casos obtiene un resultado idéntico y en un 54 % el resultado se encuentra confinado en el ratio entre 1,5 y 1.

5. Aproximación Alternativa

El algoritmo de aproximación propuesto posee un problema evidente, se suelen cerrar paquetes con capacidad desaprovechada. Existen alternativas que intentan aprovechar el espacio remanente de los envases como el **First Fit**, sin embargo su complejidad es $O(N^2)$ lo cuál lo hace poco conveniente. Pensando posibles alternativas que intenten aprovechar la capacidad remanente, pero más eficientes recordamos la historia de (*nos ponemos de pié*) el gran Carl Friedich Gauss y la suma de todos los números del 1 al 100, donde la clave del cálculo eficiente es emplear los extremos de un conjunto ordenado.

5.1. Complejidad

El algoritmo posee una etapa preliminar de ordenamiento con complejidad $O(Nlog(N))$ y luego una etapa *greedy* de $O(N)$; por lo que podemos concluir que la complejidad total del algoritmo es de

Algorithm 2 Algoritmo Nueva Aproximación

```
1: Ordenar el conjunto de mayor a menor                                   ▷ O(N log(N))
2: Se abre un envase
3: while Hayan elementos sin empaquetar do                                ▷ O(N)
4:     Obtengo el objeto más grande del conjunto
5:     if El objeto entra en el envase then
6:         Se agrega el objeto al envase y se continua con el siguiente objeto
7:     else
8:         while Queden elementos sin empaquetar y el más chico entre en el envase abierto do ▷ O(N)[1]
9:             Obtener el objeto más chico y agregarlo al envase abierto
10:        end while
11:        Se cierra el envase actual
12:        Se abre un envase nuevo y se envasa el objeto (el más grande previamente obtenido)
13:    end if
14: end while
15: Devolver los paquetes
```

$O(Nlog(N))$

5.2. Pruebas de Ejecución

Se realizaron diversas corridas de ejecución con datasets aleatorios de valores diversos, se compararon los resultados de nuestra aproximación contra la brindada por la cátedra, estos fueron los resultados obtenidos

[1] $O(N)$ es el peor de los casos, suponiendo que el **while** superior se encuentre en la iteración número i, el orden será $O(N - i)$

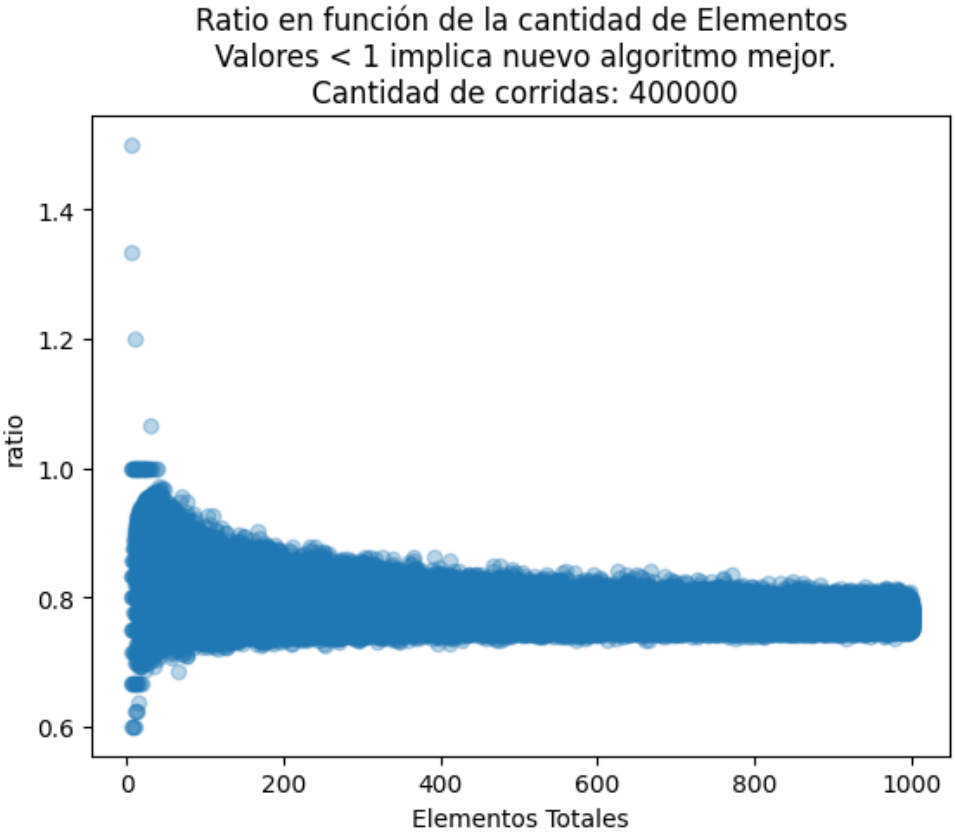


Figura 3: Ratio Nueva Aproximación vs Aproximación dada

Se observa que en la mayoría de las corridas se obtiene un resultado mejor al base

group	count	perc
$r < 0{,}5$	0	0.000000
$0{,}5 \leq r < 1$	398526	0.996315
$r = 1$	1469	0.003672
$1 < r \leq 1{,}5$	5	0.000013
$1{,}5 < r \leq 2$	0	0.000000

Cuadro 4: Porcentaje de Ratios obtenidos

asimismo el gráfico parece indicar que a mayor cantidad de elementos, mejores resultados obtiene nuestro algoritmo; verificaremos los rangos de la cantidad de elementos para asegurarnos que el rango mostrado sea uniforme.

n	n
(0, 200]	78217
(200, 400]	80354
(400, 600]	80124
(600, 800]	80553
(800, 1000]	80752

Cuadro 5: Ejecuciones totales por rango

Los valores de la tabla 5 muestran uniformidad en los rangos de las pruebas, restaría repetir las pruebas acotando la cantidad de elementos a los rangos bajos para verificar si los malos resultados encontrados en dicho rango dependen del rango o se trata de un hecho aleatorio.

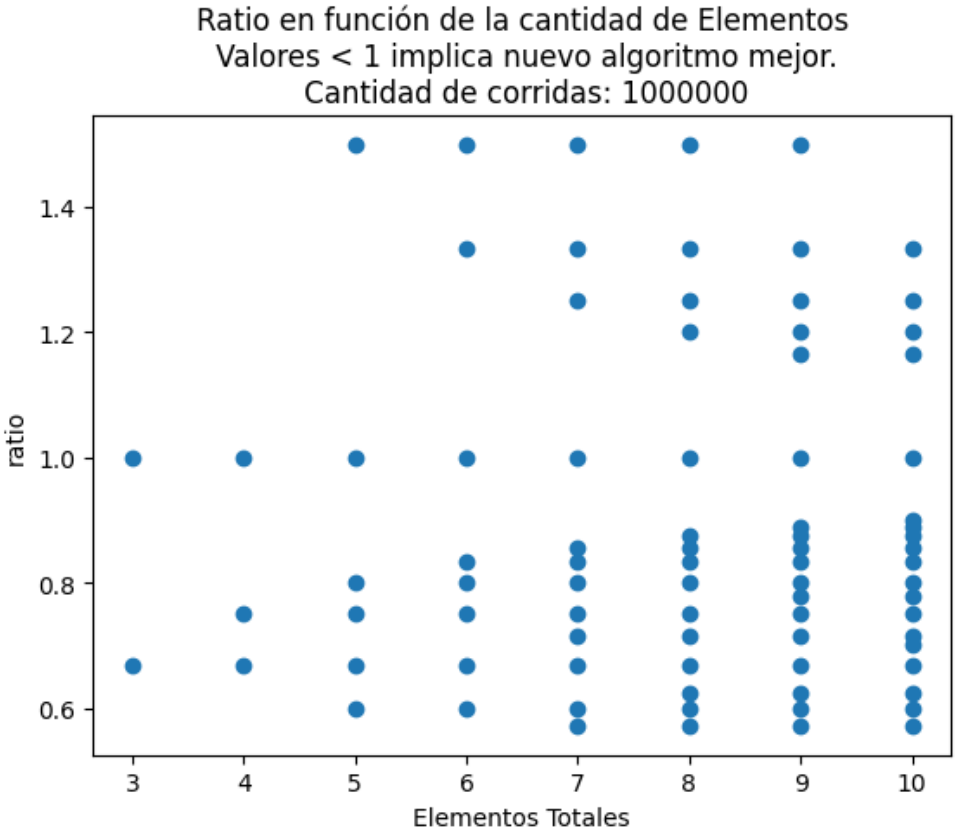


Figura 4: Ratio Nueva Aproximacion vs Aproximación dada, en rangos bajos

group	count	perc
$r < 0{,}5$	0	0.000000
$0{,}5 \leq r < 1$	394828	0.394828
$r = 1$	603489	0.603489
$1 < r \leq 1{,}5$	1683	0.001683
$1{,}5 < r \leq 2$	0	0.000000

Cuadro 6: Porcentaje de Ratios obtenidos en rangos bajos

Si se compara el cuadro 6 con el 4 observamos que en este último el porcentaje de ratios $r < 1$ es mucho mayor, mientras que en los rangos bajos predomina un $r = 1$. Sin embargo se mantiene una proporción muy minoritaria de los $r > 1$. Por lo cuál concluimos que los casos en el que nuestro algoritmo devuelve un resultado peor que el algoritmo de **Next Fit** corresponden a casos aislados no directamente relacionados a un tamaño de datos.

5.3. Comparación contra el óptimo

En la sección 4.2 obtuvimos que el algoritmo *Next Fit* a lo sumo obtiene un resultado del doble del óptimo, observando la figura 2 se observa que la mayoría de los resultados se agrupan en $r \in [1, 1{,}5]$. Adicionalmente si consideramos los resultados obtenidos en la sección 5.2, en particular los valores de los ratios de la tabla 6, resulta apreciable que la relación de los resultados del *Next Fit* con el óptimo y con nuesto algoritmo son comparables.

A continuación haremos unas pruebas para comparar nuestro algoritmo con el óptimo

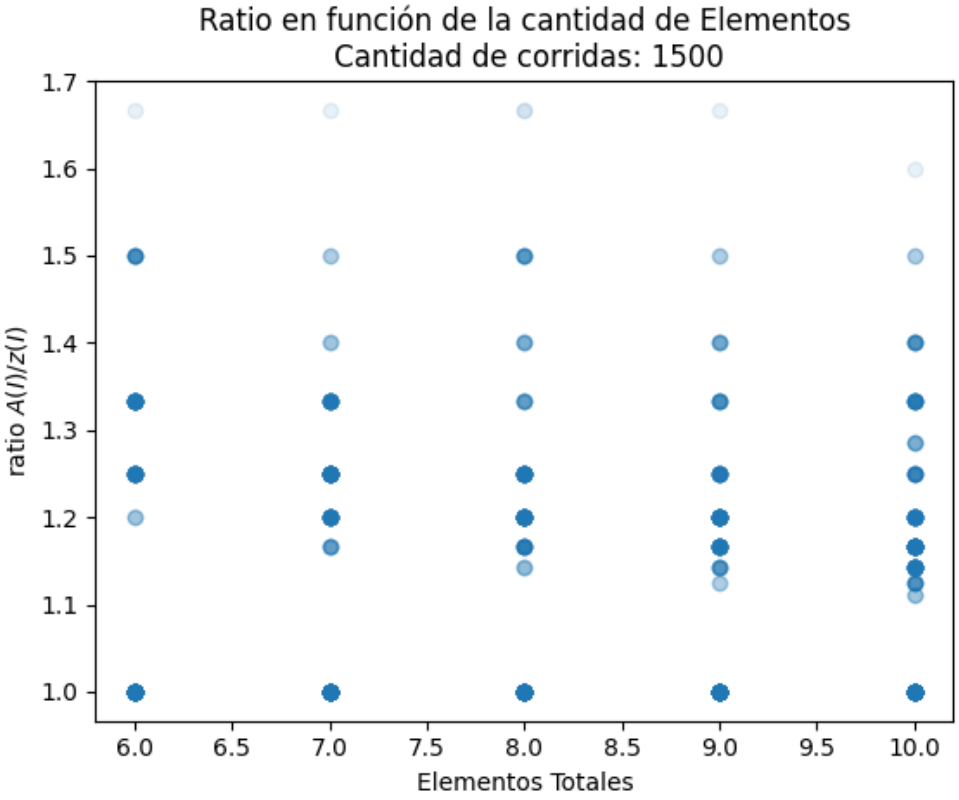


Figura 5: Ratio Nueva Aproximación vs Óptimo

group	count	perc
$r = 1$	1959	0.979500
$1 < r \leq 1{,}5$	41	0.020500
$1{,}5 < r \leq 2$	0	0.000000

Cuadro 7: Grupos Ratio Nueva Aproximación vs Óptimo

Si se comparan los resultados obtenidos en la tabla 7 con los obtenidos anteriormente en el cuadro 3, se observa que este nuevo algoritmo posee un ratio $r = 1$ en una proporción mucho mayor (97 % vs 45 %).

6. Conclusión

Durante el desarrollo de este trabajo evaluamos tres algoritmos, cada uno con sus complejidades temporales y calidad de aproximaciones

- **Fuerza Bruta**: Posee una complejidad de $O(N!)$ y su resultado es óptimo.
- **Next Fit**: Posee una complejidad de $O(N)$, con una aproximación de ratio máximo 2 y en general menor a 1.5
- **Nuestro Algoritmo**: Posee una complejidad de $O(N \cdot log(N))$, con una aproximación que en general es igual al óptimo.

si bien los resultados obtenidos resultan de pruebas acotadas dados los tiempos requeridos para obtener la solución óptima por fuerza bruta, las consideramos útiles a la hora de obtener una caracterización general de los algoritmos.

Con estos resultados, dada la necesidad de resolver un problema de empaquetado, se pueden evaluar los requerimientos y limitaciones; y con ello elegir un algoritmo acorde. Si se desea una solución exacta, sin importar el tiempo que demore y si contamos con el hardware adecuado; se puede elegir una solución mediante fuerza bruta. Caso contrario si es excluyente que se ejecute rápidamente y con menor requerimientos de hardware, el next fit. Si se busca un algoritmo con una complejidad balanceada, que en general obtenga un resultado similar al óptimo, en dicho caso será mejor nuesto algoritmo.

Todo el análisis desarrollado resulta útil no sólo para evaluar estos tres algoritmos puntuales para este problema específico, sino para entender cómo evaluar un algoritmo desde distintas ópticas y entender sus puntos fuertes y débiles.

7. Anexo

7.1. Ejecución de Script

Todas las pruebas realizadas se hicieron mediante *notebooks* y datasets generados de forma aleatoria, sin embargo acorde a lo requerido se puede invocar el script mediante

```
./tdatp2 <E>|<A>|<A2> <datos.txt> <-v|--verbose>
```

Donde poseemos dos argumentos posicionales

1. Algoritmo a Utilizar:
   - E: Emplear solución Exacta
   - A: Aproximación propuesta por el curso
   - A2: Aproximación propuesta por el grupo
2. Datos: *Path* al archivo de datos a utilizar

Opcionalmente admite el argumento `-v, --verbose` , el cual indica que además de imprimir la cantidad de envases, se debe mostrar como están constituidos los mismos

Ejemplo:

```
$ ./tdatp2 A samples/data_enunciado.txt -v

Solución Aproximada: 6
E1: {0.4}
E2: {0.8}
E3: {0.5; 0.1}
E4: {0.7}
E5: {0.6; 0.1}
E6: {0.4; 0.2; 0.2}
0.01
```

7.1.1. Samples de Prueba

En el directorio `samples` se pueden encontrar varios archivos de prueba

- data_enunciado.txt: El dataset de ejemplo dado en el enunciado
- worse_next_fit.txt: Un dataset acotado del caso que ejemplifica la cota demostrada en la sección 4.2
- random_N.txt: Datasets aleatorios de N elementos.

7.1.1.1 Random Datasets

Si se desean generar datasets aleatorios se puede invocar el archivo `data.py` de la siguiente manera

```
python data.py -n 8
8

0.84
0.95
0.38
0.41
0.12
0.75
0.21
0.47
```

eligiendo un valor de N. Si no se especifica, utiliza por defecto 5. Esto se puede *pipear* a un archivo y ser utilizado para el ejecutable del trabajo, por ejemplo

```
python data.py -n 8 | tee /tmp/aux.txt && ./tdatp2 A2 /tmp/aux.txt -v
8

0.97
0.59
0.03
0.25
0.09
0.79
0.2
0.5

Solución Aproximada Alumnos: 4
E1: {0.97; 0.03}
E2: {0.79; 0.09}
E3: {0.59; 0.2}
E4: {0.5; 0.25}
0.03
```

7.2. Notebooks de Gráficos y Pruebas

En la realización de este trabajo se realizaron diversas pruebas y graficos, los dichos fueron hechos mediante *notebooks* de Python y pueden encontrarse en el repositorio