DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP 1: Jambo-Tubos

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Miceli, Juan Pablo	424/19	micelijuanpablo@gmail.com
Zolezzi, María Victoria	222/19	zolezzivic@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

Índice

1. Introducción

En este trabajo nos dedicaremos a resolver el problema "jambo-tubos", el mismo consiste en insertar la mayor cantidad de elementos en un jambo-tubo sin superar su resistencia máxima ni aplastar los productos a apilar. Cada objeto esta provisto de un peso y una resistencia, las cuales nos permitirán notar cuando un objeto es aplastado por otro, o si superamos la capacidad máxima del tubo. La respuesta a devolver será el mayor número de objetos que podemos insertar en el tubo, cumpliendo las restricciones ya desarrolladas.

A lo largo del trabajo usaremos la siguiente notación:

- R: Resistencia del jambo-tubo.
- n: Cantidad de objetos a apilar.
- w: Vector de pesos, donde w_i representa el peso del i-ésimo producto.
- r: Vector de resistencias, donde r_i representa la resistencia del i-ésimo producto.

Para lograr una mejor comprensión del problema, expondremos algunos ejemplos junto a sus soluciones.

- Dados R = 50, n = 5, w = [15, 10, 20, 5, 8] y r = [30, 20, 10, 20, 15], la cantidad máxima de productos que podemos apilar es 4. Esto es así ya que podemos apilar los elementos 1, 2, 4 y 5. Podemos notar que la resistencia del tubo (50) no es superada por el peso de los objetos apilados (38), y que ningún elemento resultó aplastado al no haber superado las resistencias de los mismos.
- Dados R = 200, n = 5, w = [10, 15, 40, 50, 60] y r = [30, 30, 40, 50, 60], la solución es 2, insertando los elementos 1 y 2. Notar que si bien la resistencia del tubo indica que podemos apilar los 5 objetos, el peso de los elementos hace que sea imposible insertar más productos sin que estos sean aplastados.

Nuestro objetivo será utilizar diferentes técnicas para resolver el problema, analizando sus rendimientos en función de la complejidad, observando cómo varía de instancia a instancia.

Comenzaremos por explicar los métodos empleados en las secciones siguientes.

2. Fuerza Bruta

En esta técnica se busca hallar la solución generando todas las posibles soluciones del problema, y analizando cada una hasta encontrar aquella que cumpla con lo pedido, es decir hasta hallar una solución válida.

En nuestro caso, esto significa encontrar todas las distintas formas de apilar en el jambo-tubo los objetos recibidos en la cinta, para quedarnos con aquella que maximice la cantidad de objetos apilados, sin que se aplaste ningún producto, ni que se rompa el tubo.

Analizando el problema, para cada producto tenemos 2 opciones: apilarlos o no apilarlos. Teniendo esto en cuenta, podemos ver que tenemos un total de 2^n soluciones candidatas. Para generarlas, utilizaremos un algoritmo recursivo que en el paso i, dados una resistencia mínima, un vector de resistencias y uno de pesos, obtendrá la mayor cantidad de productos que se pudieron apilar considerando los productos $i+1, i+2, ..., n$, tanto teniendo en cuenta el elemento actual como descartándolo, para quedarse así con el máximo de los dos.

Como en todo algoritmo recursivo, es necesario tener un caso base. El nuestro será cuando el índice sea igual al tamaño del vector de pesos, lo que nos indica que no tenemos más productos en la cinta. Aquí, debemos decidir si la solución obtenida es válida o no. En caso de que lo sea, se devuelve cero, indicando que de ahí en adelante no se puede apilar nada más. En caso contrario, se devuelve $-\infty$, ya que este valor es neutro para el operador max.

Habiendo visto el caso base, podemos ahora ver el caso general del algoritmo. La idea será ir obteniendo soluciones parciales, e ir extendiéndolas en cada paso, primero apilando el producto actual y llamando a la función con la nueva solución parcial. Luego, desapilamos el objeto y llamamos nuevamente a la función con esta última solución parcial. Así vamos a haber obtenido la máxima cantidad de productos tanto habiendo apilado como descartado el elemento actual. Lo único que nos resta es devolver el máximo entre estos 2 valores.

Para identificar cuándo una solución es válida necesitamos ver si rompimos el jambo-tubo o si algún producto fue aplastado. Para esto, utilizamos un parámetro llamado "rMin", que nos indica cuánto peso más podemos agregar en el tubo. Este valor nos permitirá determinar, al llegar al caso base, si la solución que obtuvimos es válida o no, ya que si la resistencia mínima es menor a cero, algún elemento fue aplastado o el jambo-tubo se rompió en el proceso, con lo cual debemos descartarla. En caso de apilar un objeto, "rMin" es actualizada restándole al valor actual el peso del elemento en cuestión, y comparándolo con la resistencia del producto a apilar para quedarnos con el mínimo. En caso contrario, la resistencia no cambia.

Algorithm 1 Algoritmo de Fuerza Bruta para jambo-tubos.

```
1:  function FB(w, r, indice, rMin)
2:      if indice == n then                                   ▷ O(1)
3:          if rMin >= 0 then                                 ▷ O(1)
4:              return 0                                      ▷ O(1)
5:          else
6:              return −∞                                     ▷ O(1)
7:      nuevaRes = min(rMin − w[indice], r[indice])           ▷ O(1)
8:      maxApilando = FB(w, r, indice + 1, nuevaRes) + 1
9:      maxSinApilar = FB(w, r, indice + 1, rMin)
10:     return máx{maxSinApilar, maxApilando}.                ▷ O(1)
```

Por lo expuesto anteriormente, nuestro algoritmo genera todas las instancias posibles, por lo tanto, de existir alguna que maximice la cantidad de objetos que se pueden apilar sin aplastar ningún producto ni romper el tubo, esta va a ser visitada. Además, como paso a paso estamos retornando el máximo entre las soluciones obtenidas, también podemos afirmar que efectivamente estamos devolviendo lo que buscamos.

Para calcular la complejidad del algoritmo de Fuerza Bruta, plantearemos la ecuación de recurrencia. Para esto, debemos tener en cuenta que cada instancia del problema se divide en dos subproblemas de tamaño k-1, donde k representa el tamaño de la instancia actual, y que las operaciones de las líneas 2, 8 y 10 tienen tiempo de ejecución constante al tratarse de operaciones elementales. El caso base pertenece a $\mathcal{O}(1)$ ya que consiste en realizar dos comparaciones.

Sea k la cantidad de elementos que quedan por apilar.
La ecuación de recurrencia nos queda:

$$\text{T(k)}= \begin{cases} \mathcal{O}(1) & si\ k = 0 \\ 2T(k-1) + m & en\ caso\ contrario, donde\ m\ es\ una\ costante. \end{cases}$$

Queremos probar que $\text{T(k)} \in \mathcal{O}(2^k)$. Lo que es equivalente a probar que:

$\text{T(k)} \leq C2^k\ \forall k \in \mathbb{N}$, con $\text{C} \in \mathbb{R}$.

Lo probaremos por inducción en k.
Caso base: k=0.

T(0)= $m \leq m2^0$.
Tomando $C \geq m$, T(0) $\leq C$, y por lo tanto, el caso base es verdadero.

Paso inductivo: suponemos que T(k')$\leq 2^{k'}C'$ vale $\forall k' < k$ para alguna constante C'.
Queremos ver que T(k') $\Rightarrow$ T(k).
T(k)=$2T(k-1)+m$, y por hipótesis inductiva,

$$T(k) \leq 2(2^{k-1}C') + m = 2^kC' + m.$$

Tomando $C \geq C' + m$ se cumple la desigualdad y por lo tanto concluímos que el algoritmo $FB \in \mathcal{O}(2^k)$.

3. Backtracking

Esta técnica es muy similar a Fuerza Bruta (sección 2) ya que en un principio intentará recorrer todas las soluciones posibles, pero en este caso usaremos la información que podemos extraer del planteo del problema para evitar continuar generando soluciones parciales que no derivarán en una solución válida u óptima.

Para esto, introducimos el concepto de *poda*. La misma describe una condición por la cual debemos dejar de extender la solución parcial actual. Existen dos tipos de podas: las podas por factibilidad y las podas por optimalidad.

- Poda por factibilidad: consiste en dejar de extender soluciones parciales que no tienen ninguna posibilidad de derivar en un solución válida. En nuestro caso, sabemos que si por ejemplo, el primer objeto que apilamos rompe el jambo-tubo, no seguiremos generando soluciones que contengan este elemento, ya que sabemos que no serán válidas. Luego, deducimos que podemos prescindir de todas las soluciones parciales en las que se rompa el tubo o se aplaste algún producto. Esta poda puede encontrarse en la línea 2 del pseudocódigo 2.

- Poda por optimalidad: en este caso, vamos a dejar de generar soluciones que podrían ser válidas porque sabemos que no se convertirán en soluciones óptimas. Por ejemplo, si ya sabemos que la máxima cantidad de objetos que pudimos meter en el jambo-tubo fue 10, y en la solución parcial que estamos extendiendo apilamos 3 objetos y solo nos restan 6 productos en la cinta, aunque pudiésemos apilar todos esos elementos, no superaremos al máximo ya obtenido. Por esta razón, podemos prescindir de esa rama. En general, prescindiremos de todas las soluciones parciales que no tengan posibilidades de superar, aunque se apilen todos los productos faltantes, a la cantidad máxima de objetos apilados conseguida hasta el momento. Esta poda puede verse en la línea 4 del algoritmo 2.

Para implementar este algoritmo, usaremos como base el algoritmo 1 visto en la sección 2; al principio del mismo se agregararán las 2 podas. Para aplicar la poda de factibilidad, basta con chequear que el parámetro "rMin" no sea negativo, ya que esto indicaría que algún objeto fue aplastado o que se superó la capacidad máxima del jambo-tubo. Para la poda de optimalidad vamos a incorporar dos nuevos parámetros llamados "maxActual" y "*#apilados*". El primero indica cuál fue la cantidad máxima de productos que pudimos apilar exitosamente hasta el momento; este solo es actualizado cuando obtenemos una solución válida (notar que este parámetro debe ser pasado por referencia para que su actualización sea efectiva). El segundo nos permite saber en cada solución parcial cuántos son los elementos que se encuentran apilados; esto lo usaremos para actualizar el *maxActual* y para identificar la cantidad máxima de productos que se pueden guardar si guardaran todos los restantes. De esta forma, podemos compararla contra la mejor solución obtenida hasta el momento (*maxActual*), y en caso de que la cantidad de productos apilados sea menor o igual, descartamos toda esa rama.

El resto del algoritmo se comporta como el algoritmo 1.

Algorithm 2 Algoritmo de Bactracking para jambo-tubos.

```
1: function BT(w, r, indice, rMin, maxActual, #apilados)
2:     if rMin < 0 then                                   ▷ Poda por factibilidad O(1)
3:         return −∞                                                          ▷ O(1)
4:     apilables = (n − indice) + #apilados                                   ▷ O(1)
5:     if apilables <= maxActual then                     ▷ Poda por optimalidad O(1)
6:         return −∞                                                          ▷ O(1)
7:     if indice == n then                                                    ▷ O(1)
8:         if rMin < 0 then                                                   ▷ O(1)
9:             return −∞                                                      ▷ O(1)
10:        maxActual = max(maxActual, #apilados)                              ▷ O(1)
11:        return 0                                                           ▷ O(1)
12:    nuevaRes = min(rMin − w[indice], r[indice])                            ▷ O(1)
13:    maxApilando = BT(w, r, indice + 1, nuevaRes, maxActual, #apilados + 1) + 1
14:    maxSinApilar = BT(w, r, indice + 1, rMin, maxActual, #apilados)
15:    return máx{maxSinApilar, maxApilando}.                                 ▷ O(1)
```

Analizando la complejidad del algoritmo, podemos ver que en peor caso (cuando no podemos podar en ningún paso), la complejidad es igual a la del algoritmo de Fuerza Bruta ($\mathcal{O}(2^n)$), ya que las podas introducidas y la actualización de *maxActual* tienen costo constante. Sin embargo, la introducción de estas podas nos permite en muchos casos reducir la cantidad de llamados recursivos, disminuyendo la complejidad del algoritmo. Por ejemplo, si todos los productos de la cinta tienen un peso mayor a la resistencia del jambo-tubo, en ningún momento vamos a evaluar la posibilidad de apilar un producto, haciéndo que el algoritmo tenga complejidad lineal.

4. Programación Dinámica

Esta técnica consiste en dividir el problema en subproblemas más pequeños, que son más sencillos de resolver, pero a diferencia de un algoritmo recursivo común, cada resultado obtenido es almacenado. De esta forma, evitamos resolver más de una vez una instancia.

Podemos notar que utilizar este tipo de algoritmos no siempre vale la pena, ya que para casos en los que no tenemos que llamar a la función con la misma instancia en repetidas ocasiones, almacenar los resultados es verdaderamente una pérdida de tiempo y espacio. Por esta razón introducimos el concepto de *superposición de subproblemas*. El mismo nos permite distinguir cuando un problema puede ser resuelto eficientemente con un algoritmo de Programación Dinámica, mediante la comparación entre la cantidad de subproblemas distintos y la cantidad de llamados recursivos que requiere. Lo que buscamos es que el primero sea mucho menor que el segundo.

Como mencionamos, en estos algoritmos deben almacenarse los resultados obtenidos, para ello se utiliza una estructura de memoización. En nuestro caso, utilizaremos una matriz *memo*, donde $memo[R][i]$ contiene la mayor cantidad de elementos que pueden apilarse en un jambo-tubo de resistencia R, si se tienen los últimos i productos en la cinta. Esta matriz será inicializada con -1, ya que esa nunca puede ser una solución válida.

Veamos el algoritmo que planteamos para confirmar que esta propiedad se cumple.

Algorithm 3 Algoritmo de Programación Dinámica para jambo-tubos.

```
1:  function PD(memo, w, r, indice, rMin)
2:      if rMin < 0 then                                              ▷ O(1)
3:          return −∞                                                 ▷ O(1)
4:      if indice == n or rMin == 0 then                              ▷ O(1)
5:          memo[rMin][indice] = 0                                    ▷ O(1)
6:          return 0                                                  ▷ O(1)
7:      if memo[rMin][indice] == −1 then                              ▷ O(1)
8:          nuevaRes = min(rMin − w[indice], r[indice])               ▷ O(1)
9:          maxApilando = PD(memo, w, r, indice + 1, nuevaRes) + 1
10:         maxSinApilar = PD(memo, w, r, indice + 1, rMin)
11:         memo[rMin][indice] = max(maxSinApilar, maxApilando)       ▷ O(1)
12:     return memo[rMin][indice]                                     ▷ O(1)
```

Podemos ver que este algoritmo hace esencialmente lo mismo que hacía Backtracking(2), exceptuando que en este caso estamos guardando los resultados obtenidos para cada instancia en nuestra matriz de memoización, con el fin de reducir los cálculos repetidos.

Como vimos anteriormente, la cantidad de llamados recursivos que debemos resolver es 2^n, mientras que la cantidad de subproblemas recursivos es $n * R$, ya que en nuestra función solo tenemos dos parámetros que no son fijos. Luego, la cantidad de subproblemas es equivalente a la cantidad de instancias posibles que tenemos, es decir, a la cantidad de combinaciones entre ellos.

Vemos entonces que la propiedad de superposición de subproblemas solo se cumple si $n * R << 2^n$, lo cual puede no ocurrir siempre. En caso de que no sea así, sería una mejor opción encarar el problema con un algoritmo de Backtracking, ya que su complejidad temporal y espacial será mejor.

5. Experimentación

En esta sección expondremos los experimentos que fuimos realizando con el fin de analizar empíricamente los algoritmos propuestos, buscando contrastar los resultados obtenidos con los teorizados en las secciones anteriores.

5.1. Métodos

Los métodos que usaremos durante la experimentación son los siguientes:

- FB: Algoritmo 1 de Fuerza Bruta mostrado en la sección 2.
- BT: Algoritmo 2 de Backtracking mostrado en la sección 3.
- BT_F: Algoritmo 2 de Backtracking mostrado en la sección 3, aplicando únicamente la poda de factibilidad, es decir, la que se muestra en la linea 2.
- BT_O: Algoritmo 2 de Backtracking mostrado en la sección 3, aplicando únicamente la poda de optimalidad, es decir, la que se muestra en la linea 4.
- PD: Algoritmo **??** de Programación Dinámica mostrado en la sección 4.

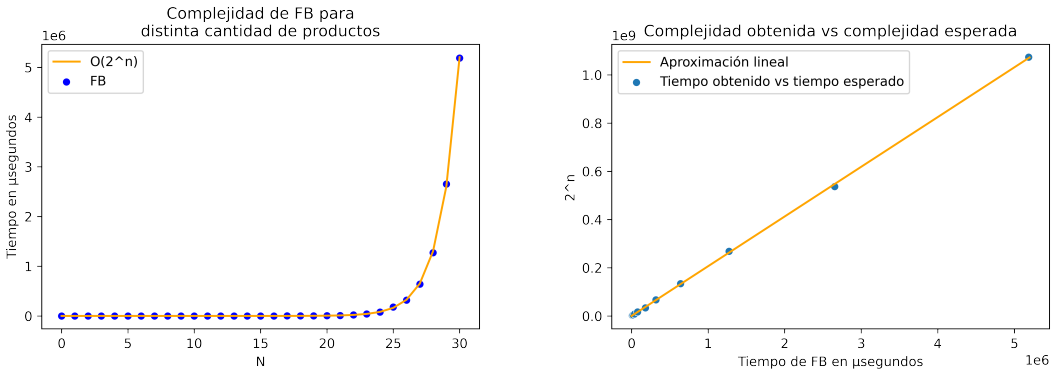
5.2. Experimento 1: Complejidad de Fuerza Bruta

Vamos a analizar la complejidad del algoritmo con el objetivo de confirmar nuestra hipótesis mencionada en la sección 2, es decir, que la complejidad pertenece a $\mathcal{O}(2^n)$.

Para esto, generaremos una familia de instancias en la cual los pesos y resistencias de los productos varían aleatoreamente entre 4 y 10, y entre 10 y 50 respectivamente. Luego, variaremos la cantidad de productos para ver qué impacto tiene este parámetro en la complejidad del algoritmo.

Esperamos ver que existe una relación lineal entre los tiempos medidos y la complejidad teórica.

Comenzaremos por graficar los tiempos obtenidos en función de la cantidad de productos para una resistencia del jambo-tubo igual a 20; notar que no variaremos el valor de este parámetro ya que este no tiene ninguna incidencia en la complidad del algoritmo.



(a) Complejidad esperada vs tiempos de ejecución obtenido.

(b) Correlación entre la complejidad esperada y los tiempos de ejecución obtenidos.

Figura 1: Análisis gráfico de la complejidad de FB.

Observando el gráfico 1a podemos ver que los tiempos obtenidos se pueden aproximar por la curva dada por la función:

$$f(x) = 4{,}55 \times 10^{-6} \times 2^n,$$

lo cual indicaría que la cantidad de productos tiene una relación exponencial con el tiempo de ejecución. Para confirmar esto, calculamos el coeficiente de Pearson entre el tiempo de ejecución obtenido y el esperado, el mismo tiene un valor de 0.999992. Lo cual nos muestra que estas dos variables están fuertemente relacionadas de manera lineal. Por último, veamos esta relación lineal de manera gráfica en la figura 1b.

5.3. Experimento 2: Complejidad de Backtracking

Queremos ahora analizar la complejidad de backtracking. Nuestra hipótesis sobre este algoritmo es que, en un caso promedio, gracias a las podas debería funcionar considerablemente mejor que Fuerza Bruta. Por otra parte, creemos que en peor caso (cuando no podamos aplicar las podas) el algoritmo se comportará como el de Fuerza Bruta (1).

Para realizar este experimento queríamos correr el algoritmo con instancias que constituyan su peor caso, sin embargo, como desarrollaremos en la sección 5.4, nos dimos cuenta que el peor caso para una de las podas era el mejor caso para la otra, es decir que no hallamos instancias para las que ninguna poda sea efectiva. Por esta razón, decidimos correr el experimento con una serie diversa de instancias, en las que iremos variando la resistencia del jambo-tubo así como la relación entre los pesos y las resistencias de los objetos. Definiremos la relación entre los pesos y las resistencias como:

$$Rel(w, r) = \frac{E(w)}{E(r)}$$

.

Siendo r y w las variables aleatorias que generarán las resistencias y los pesos de los productos, y $E(x)$ la esperanza de la variable aleatoria x.

Vamos a analizar 3 valores para esta relación:

- $Rel(w, r) = 1$: ambas variables aleatorias son generadas mediante una distribución uniforme continua $\mathcal{U}(30, 60)$. Obteniendo así que $Rel(w, r) = \frac{45}{45} = 1$.
- $Rel(w, r) = \frac{1}{2}$: la variable de pesos es generada con la misma distribución que en el ítem anterior mientras que la variable de resistencias es generada con una distribución continua $\mathcal{U}(75, 105)$.
- $Rel(w, r) = 2$: aquí invertimos las variables uniformes continuas usadas en el item anterior.

En la siguiente figura, expondremos los resultados de este experimento.

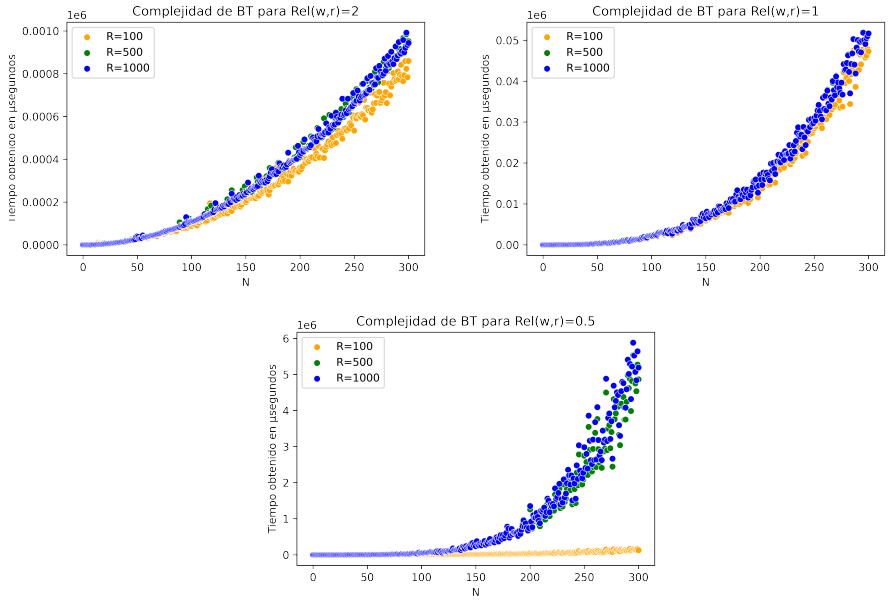


Figura 2: Análisis de complejidad del método BT para distintas relaciones w, r y distintas resistencias del jambo-tubo.

Como podemos ver en la figura 2, los tiempos de ejecución varían mucho en cada caso. En particular, notamos una muy buena performance cuando corremos el algoritmo con instancias en las que $Rel(w, r) = 2$. Esto se debe a que prácticamente todo segundo objeto que queramos apilar terMina aplastando al objeto ya apilado, lo cual implica que la poda de factibilidad va a descartar muchas soluciones parciales, disminuyendo en gran medida la cantidad de llamados recursivos.

En el caso donde trabajamos con instancias en las que $Rel(w, r) = 1$, notamos un decremento en la performance, debido a que se aumenta la probabilidad de poder apilar más de 2 objetos. Esto provoca que la poda de factibilidad no sea tan efectiva. Por otra parte, al estar apilando un número tan reducido de objetos en comparación con la cantidad total de productos, la poda de optimalidad no juega un papel decisivo.

Por último, en el caso donde $Rel(w, r) = 0{,}5$, nos encontramos 2 comportamientos muy distintos. El primero se da con $R = 100$, aquí pasa algo similar a lo visto en el caso anterior, esta vez podremos apilar a lo sumo 3 objetos antes de romper el jambo-tubo. En cambio, para los valores más grandes de R, podemos llegar a apilar 4 objetos, lo cual a priori no parece un cambio significativo, sin embargo debemos notar que la cantidad de maneras de elegir 3 productos de entre 300 es muchísimo menor que si contáramos con 4, siendo el primer valor 4,455,100 y el segundo 330,791,175. Aquí es donde nos percatamos de la importancia de la poda de factibilidad. Por otra parte, esta cantidad de elementos aún resulta insustancial para que la poda de optimalidad logre reducir de manera significativa la cantidad de llamados recursivos que deben realizarse.

Más allá de lo expuesto en los últimos párrafos, si comparamos estos gráficos con el gráfico de la figura 1a podemos notar que la complejidad de backtracking en todos los casos analizados es

notablemente mejor que la de Fuerza Bruta. Es por esto que concluímos que las podas constituyen un avance significativo en la performance del algoritmo.

Dado que en ninguno de los tres casos la poda de optimalidad tuvo un gran impacto en la complejidad del algoritmo, estudiaremos con más detalle la efectividad de cada poda por separado.

5.4. Experimento 3: Efectividad de las podas

Como mencionamos anteriormente, la complejidad de este algoritmo depende fuertemente de la aplicación de las podas. Es por esto, que ahora analizaremos el rendimiento de cada una de ellas por separado. Haciendo un análisis del funcionamiento de las mismas, podemos encontrar un mejor y peor caso para cada una.

En el caso de la poda de factibilidad, el peor caso sería que todos los objetos puedan ser apilados. Esto haría que $rMin$ sea positivo en todo momento, por lo cual la poda nunca se aplicaría. Por lo tanto el algoritmo deberá extender todas las soluciones parciales y se comportará igual que fuerza bruta (recordar que estamos considerando que no contamos con la poda de optimalidad).

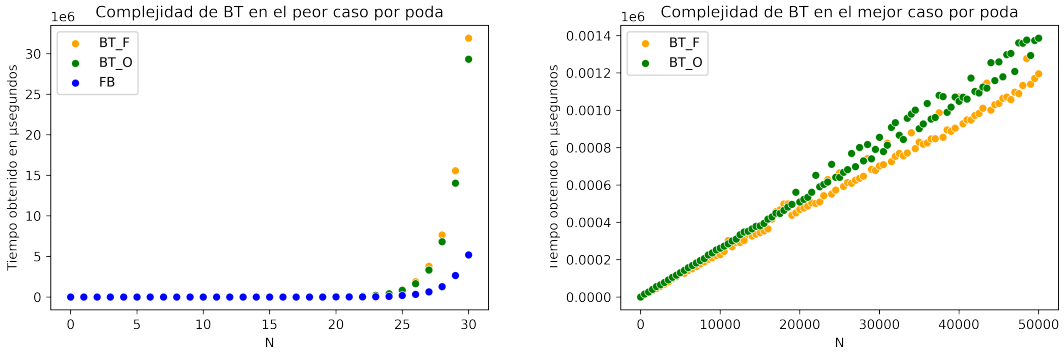
En cambio, si ningún producto puede ser colocado en el jambo-tubo, la poda se ejecutará en todos los llamados recursivos, y solo deberemos analizar una rama del árbol de recursión, constituyendo así el mejor caso de esta versión. Por esta razón esperamos que el algoritmo tenga una complejidad lineal en función de la cantidad de productos en la cinta.

Veamos ahora la poda de optimalidad: en caso de que no podamos meter ningún elemento, *maxActual* será 0 y *apilables* será mayor a este último para toda solución parcial, obteniendo así el peor caso de esta versión del algoritmo. Es por esto que creemos que la complejidad temporal de backtracking en este caso será la misma que la del algoritmo 1 de fuerza bruta.

El mejor caso para esta versión será cuando podamos apilar todos los productos, ya que la primer rama del árbol de recursión que se recorre es aquella en la que se apilan todos los objetos. Luego de calcularla, no existe ninguna otra solución que pueda superarla, y por ello no serán tenidas en cuenta. Dicho esto, conjeturamos que la complejidad temporal en este caso es lineal.

Las instacias a utilizar tendrán un jambo-tubo de resistencia 1000 y sus productos serán generados de la siguiente manera:

- Mejor caso factibilidad/peor caso optimalidad: todos los objetos pesan infinito y tienen resistencia 1.
- Mejor caso optimalidad/peor caso factibilidad: todos los objetos pesan 1 y tienen resistencia infinito.



(a) Tiempos obtenidos ejecutando el peor caso de BT_F y BT_O, comparándolos contra FB.

(b) Tiempos obtenidos ejecutando el mejor caso de BT_F y BT_O.

Figura 3: Análisis de complejidad del método BT para mejor y peor caso de BT_F y BT_O.

Mirando la figura 3a notamos que al ejecutar los algoritmos en su peor caso obtenemos una complejidad mayor a la obtenida en Fuerza Bruta. Sabemos que la cantidad de llamados recursivos

es la misma para los 3 algoritmos analizados, por lo que creemos que la diferencia en los tiempos obtenidos entre los algoritmos de Backtracking y el algoritmo de Fuerza Bruta se debe al chequeo que se debe realizar para identificar si corresponde podar o no en cada paso. Es por esto que concluímos que los 3 algoritmos tienen la misma complejidad, como teorizamos al principio de este experimento, sin embargo es pertinente aclarar que si no se puede ejecutar ninguna poda lo más seguro es que el algoritmo más performante sea el de Fuerza Bruta, ya que es el que realiza menos operaciones elementales.

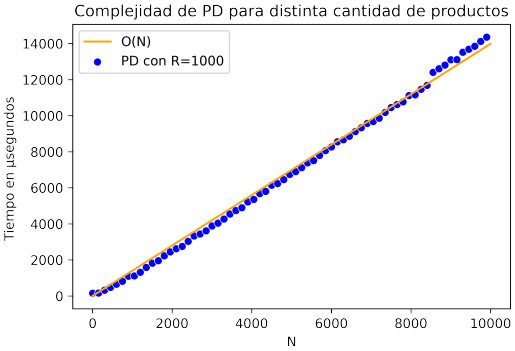
Si ahora analizamos la figura 3b podemos ver que tenemos dos algoritmos fuertemente lineales, confirmando así la hipótesis planteada.

5.5. Experimento 4: Complejidad de Programación Dinámica

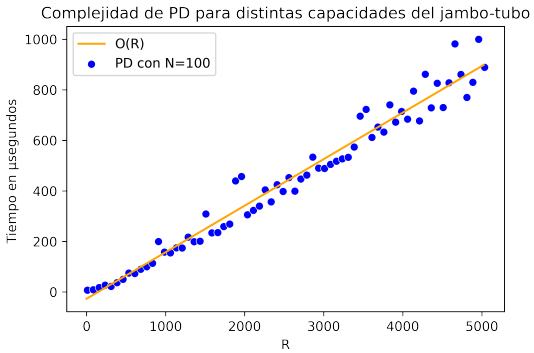
Para este experimento utilizaremos una familia de instancias en la que los pesos y las resistencias de los productos varían aleatoriamente entre 4 y 10, y 5 y 50 respectivamente. Notar que en este caso no variaremos la relación entre estos parámetros como si lo hicimos en la sección 3, ya que la complejidad de nuestro algoritmo no depende de la misma.

Comenzaremos por graficar los tiempos obtenidos dejando R fijo y variando el N. Esperamos ver que el crecimiento de los tiempos se de en forma lineal, ya que la hipótesis planteada en la sección 4 sostenía que la complejidad del algoritmo pertenecía a $\mathcal{O}(N * R)$. Estos resultados se muestran en la figura 4a.

Así mismo, repetiremos el experimento anterior dejando N fijo y variando R. Nuevamente esperamos ver un crecimiento lineal por lo expuesto en el párrafo anterior. Expondremos los resultados en la figura 4b.



(a) Tiempos obtenidos ejecutando PD en instancias con distinta cantidad de productos e igual resistencia comparados contra la complejidad teórica.

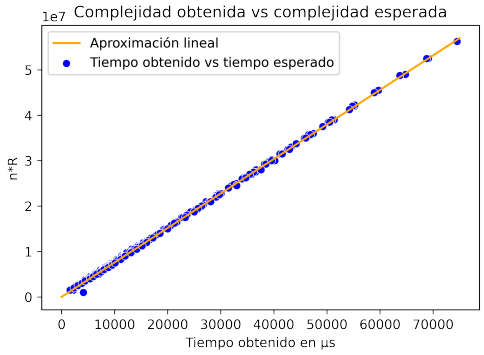


(b) Tiempos obtenidos ejecutando PD en instancias con distinta resistencia e igual cantidad de productos comparados contra la complejidad teórica.

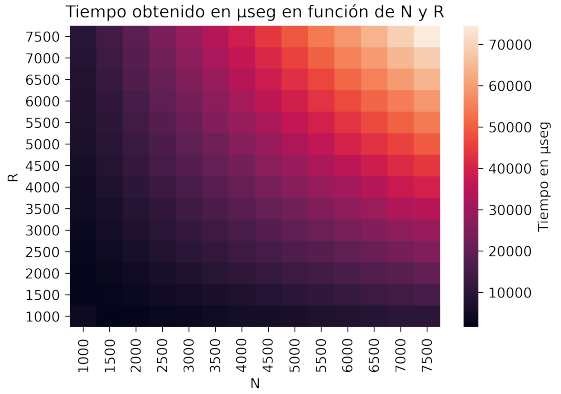
Viendo la figura 4a, notamos que se ve un crecimiento lineal muy marcado, mientras que en la 4b, este crecimiento también aparece, pero de forma más ruidosa.

Continuando con la búsqueda de pruebas para confirmar la hipótesis planteada, analizaremos la complejidad del algoritmo variando tanto la resistencia del jambo-tubo como la cantidad de productos a apilar. Estos van se moverán uniformemente entre 1000 y 7500 con intervalos de 500.

Veamos los resultados del experimento anterior en las figuras 5a y 5b.



(a) Correlación entre la complejidad esperada y los tiempos de ejecución obtenidos.



(b) Tiempos obtenidos ejecutando PD en función de la cantidad de productos y la resistencia del jambo-tubo.

Observando la figura 5a y teniendo en cuenta que la correlación de Pearson entre el tiempo obtenido en la práctia y la complejidad teórica es igual a 0,9998, confirmamos que nuestro algoritmo pertenece a $\mathcal{O}(N * R)$. Además, el heatmap de la figura 5b nos muestra que hay un crecimiento muy similar en el tiempo de ejecución, tanto cuando se aumenta R como cuando se aumenta N. Eso se puede ver viendo el patrón simétrico que se forma si se traza una diagonal desde el extremo inferior izquierdo hacia el extremo superior derecho.

6. Conclusiones

A lo largo del trabajo hemos analizado el funcionamiento de los 3 algoritmos planteados para el problema del jambo-tubo. Lo primero que notamos es que Fuerza Bruta es fácil de implementar, ya que no es necesario realizar un gran estudio del problema para obtener los resultados buscados; sin embargo, tiene una muy mala complejidad al ser un algoritmo exponencial, lo cual nos hace casi imposible trabajar con instancias grandes. En el caso de Backtracking, destacamos que nos otorga una gran mejora con respecto al anterior, sin necesidad de modificar tanto el código, ya que únicamente debemos agregar las podas, que terminan siendo lo más complejo de idear. Consideramos una contra el hecho de que haya una brecha tan grande entre la complejidad del peor y del mejor caso, a causa de que dificulta la estimación de la complejidad en un caso promedio. Por último, consideramos que PD es muy útil si se cumple la propiedad de superposición de subproblemas. En caso de que no sea así, se termina obteniendo la misma complejidad que podemos conseguir ejecutando Fuerza Bruta o Backtracking, pero consumiendo muchísimo espacio. En contraposición a la contra mencionada de Backtracking, creemos que es muy positivo tener un orden de complejidad preciso para poder evaluar fácilmente si es conveniente utilizar esta técnica.