DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 1: Optimizando Jambo-tubos

2021-1C Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Dacunda Ratti, Jerónimo	710/18	jero.d.r22@gmail.com
Alonso Rehor, Ignacio	195/18	arehor.ignacio@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

1. Introducción

La cadena de supermercados Jambo desea ofrecer un sistema de empacado que reemplace a las bolsas plásticas por los denominados Jambo-tubos. Dicho sistema consiste de un robot que recibe los productos uno a uno por medio de una cinta transportadora, y para cada producto debe decidir si lo agrega o no al tubo. Los productos agregados al tubo se van apilando uno encima de otro. Su objetivo es, conociendo el peso y la resistencia de cada producto (es decir, el peso que puede soportar), el de introducir en el tubo la cantidad máxima de productos, sin aplastar ninguno de los productos previamente colocados y sin que se rompa el tubo. Formalmente, sea $\{s_k\}_{1 \leq k \leq n}$ una sucesión de $n \geq 0$ productos, con $s_k = (w_k, r_k)$ donde w es el peso del producto y r su resistencia. Dado un **Jambo-tubo** de resistencia $\mathcal{R} \in \mathbb{N}$, el problema consiste en hallar la longitud de la mayor subsucesión $\{s_{k_r}\}_{1 \leq k_r \leq n'} \subseteq \{s_k\}$ tal que

$$\sum_{r=1}^{n'} s_{k_r}(0) = \sum_{s_i \in \{s_{k_r}\}} w_i \leq \mathcal{R}.$$

Y también tiene que ser

$$\sum_{i=m}^{n'} w_{k_i} \leq r_{k_j}, \quad \text{para todo } 1 \leq j < m \leq n',$$

es decir que los productos no se pueden romper entre si.

A continuación se exhiben algunos ejemplos con sus correspondientes respuestas esperadas. Sea

$$s_k = (10, 45), (20, 8), (30, 15), (10, 2), (15, 30),$$

y $\mathcal{R} = 50$ entonces existen $n+1 = 6$ respuestas posibles provenientes de 2^5 soluciones, de las cuales 13 son factibles, como por ejemplo: $s_k^1 = s_1, s_2$; $s_k^2 = s_1, s_3, s_4$; $s_k^3 = s_2$; y $s_k^4 = ()$. La de mayor longitud es s_k^2, siendo $|s_k^2| = 3$, y no existe ninguna solución factible que incluya más elementos, por lo tanto la respuesta en este caso es 3. Notar, por ejemplo, que si se agrega el producto s_2 al tubo, no se puede agregar ningún producto de los subsiguientes dado que todos poseen un peso mayor a r_2. Por otra parte, dada la misma secuencia $\{s_k\}$ y una resistencia $\mathcal{R} = 9$, cualquier producto que se agregue rompería el tubo, por lo tanto la respuesta es 0.

El objetivo de este trabajo es abordar el problema de optimización de Jambo-tubos utilizando tres técnicas de programación distintas y evaluar la efectividad de cada una de ellas para diferentes conjuntos de instancias. En primer lugar se utiliza *fuerza bruta* (FB o BF) que consiste en enumerar todas las posibles soluciones, de manera recursiva, buscando aquellas que son factibles. Luego, se introducen podas para reducir el número de nodos de este árbol recursivo en busca de un algoritmo más eficiente, obteniendo un algoritmo de *backtracking* (BT). La ultima técnica que vamos a abordar consiste en utilizar una estructura para almacenar resultados de subproblemas previamente calculados. A este proceso se lo conoce como *memoización* y a esta técnica como *programación dinámica* (PD o DP).

En las siguientes secciones vamos a abordar cada una de las técnicas con más detalle.

El trabajo va a estar ordenado de la siguiente manera: primero en la Sección 2 se define el algoritmo recursivo de Fuerza Bruta para recorrer todo el conjunto de soluciones y se analiza su complejidad. Más tarde, en la Sección 3 se explica el algoritmo de Backtracking con un breve análisis de mejores y peores casos. Luego, se introduce el algoritmo de DP en la Sección 4 junto con la demostración correspondiente de correctitud y un análisis de complejidad. Finalmente, en la Sección 5 se presentan los experimentos computacionales con sus respectiva discusión, y las conclusiones finales se encuentran en la Sección 6.

2. Fuerza Bruta

Un algoritmo de Fuerza Bruta enumera todo el conjunto de soluciones en búsqueda de aquellas factibles u óptimas según si el problema es de decisión u optimización. En este caso, el conjunto de soluciones está compuesto por todas las subsucesiones de s_k.

Por ejemplo, si $s_k = (10, 5), (8, 10), (7, 20)$ y $\mathcal{R} = 30$, las soluciones posibles son

a) $()$ b) $(10, 5)$ c) $(8, 10)$

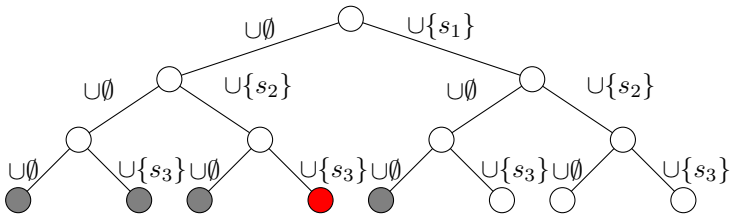


Figura 1: Ejemplo de ejecución del Algoritmo 1 para $s_k = (10,5),(8,10),(7,20)$ y $\mathcal{R} = 30$.
En rojo la solución óptima $s^{opt} = (8,10),(7,20)$ y en gris las otras soluciones factibles.

d) $(7,20)$ e) $(10,5),(8,10)$ f) $(10,5),(7,20)$

g) $(8,10),(7,20)$ h) $(10,5),(8,10),(7,20)$

De las cuales a), b), c), d) y g) son factibles. Dentro de las factibles, la de mayor longitud es g) por lo tanto la respuesta es 2.

La idea del Algoritmo 1 para resolver el Jambo-tubo es ir generando las soluciones de manera recursiva decidiendo en cada paso si un elemento de $\{s_k\}$ es considerado o no y quedándose con la mejor solución de alguna de las dos ramas. Finalmente, al identificar una solución, determinar si es factible y de ser así, devolver el cardinal de esa solución.

En la Figura 1 se ve un ejemplo del árbol de recursión para la instancia $s_k = (10,5),(8,10),(7,20)$ y $\mathcal{R} = 30$. Cada nodo intermedio del árbol representa una *solución parcial*, es decir, cuando aún no se tomaron todas las decisiones de qué elementos incluir, mientras que las hojas representan a todas las soluciones (8 en este caso). La solución óptima está marcada en rojo y las otras soluciones factibles en gris. Notar que la solución al problema original es exactamente `jt_fb(0,`$\mathcal{R}, \{s_k\}$`)`.

Algorithm 1 Resolución de Jambo-tubo por Fuerza Bruta

```
1: function JT_FB(i, r, S)
2:     if i = n then
3:         if r ≥ 0 then return 0 else return −∞
4:     min ← mín(r − w_i, r_i)
5:     return máx{1 + JT_FB(i + 1, min, S), JT_FB(i + 1, r, S)}.
```

Observación: se hace referencia w_i y r_i como el primer y segundo componente, respectivamente, de la tupla $s_i \in S$.

La correctitud del algoritmo se basa en el hecho de que se generan todas las posibles soluciones, dado que para cada elemento de $\{s_k\}$ se obtienen dos ramas, una considerándolo en el conjunto y la otra dejándolo de lado. Al haber generado todas las posibles soluciones, debe encontrarse la óptima, es decir la mejor de todas ellas.

La complejidad del Algoritmo 1 para el peor caso es $O(2^n)$. Esto es así, porque el árbol de recursión es un árbol binario completo de $n+1$ niveles (contando la raíz), dado que cada nodo se ramifica en dos hijos y en cada paso el parámetro i es incrementado en 1 hasta llegar a n. Además, es importante observar que la solución de cada llamado recursivo toma tiempo constante dado que las lineas 2, 3, y 4 solamente hacen operaciones elementales como restas, mínimos y comparaciones. Como corolario, se puede concluir que el algoritmo se comporta de igual manera frente a todos los tipos de instancia, dado que siempre genera el mismo número de nodos. Dicho de otro modo, el conjunto de instancias de peor caso es igual al conjunto de instancias de mejor caso.

3. Backtracking

Los algoritmos de Backtracking siguen una idea similar a Fuerza Bruta pero con algunas consideraciones especiales. En esencia, se enumeran todas las soluciones formando un *árbol de backtracking* de manera similar a Fuerza Bruta, donde en cada nodo se generan todas las posibles decisiones locales y se mantiene la mejor solución hallada con alguna de ellas. La diferencia radica en las denominadas *podas* que son reglas que permiten evitar explorar partes del árbol en las que se *sabe* que no va a existir ninguna solución de interés. Generalmente estas podas dependen de cada problema en particular, pero las más comunes suelen dividirse en dos categorías: *factibilidad* y *optimalidad*.

Antes de explicar las diferentes podas, hay que establecer ciertas definiciones: Sea $\{s_k\}$ la sucesión de productos de problema, decimos que una subsucesión $\{s_{k_r}\} \subseteq \{s_k\}$ es una *solución parcial* del problema de Jambo-Tubos si esta se puede *extender* a una solución candidata, donde extender significa considerar productos todavía no vistos. Al intentar formalizar esta definición, una primera idea podría ser definir el conjunto de soluciones parciales como toda subsucesión de $\{s_k\}$ tal que su cardinal sea menor a n, la cantidad de productos presentes en $\{s_k\}$. No obstante, esta definición implica que toda solución candidata del problema tiene todos los productos, y por lo tanto esta definición no es correcta. Para solucionar esta ambigüedad vamos codificar una solución parcial como un vector $v \in \mathbb{R}^m$ con $m < n$, dado por

$$v = (a_1, \ldots, a_m), \quad \text{con } a_i \in \{1, 0\}.$$

Donde

$$a_i = \begin{cases} 1 & \text{si } s_i \text{ pertenece a la solución parcial,} \\ 0 & \text{si } s_i \text{ no pertenece a la solución parcial.} \end{cases}$$

De esta manera podemos describir inequívocamente las posibles soluciones parciales (y candidatas) del problema. Notemos que, bajo esta codificación, un vector $v \in \mathbb{R}^m$ representa una solución candidata si y solo si $m = n$.

Poda por factibilidad En este caso, una poda por factibilidad es la siguiente. Sea $v \in \mathbb{R}^m$ la codificación de una solución parcial, y sea r la *resistencia mínima parcial* dada por

$$r = \text{mín}\Big\{\mathcal{R} - \sum_{i=1}^{m} a_i w_i,\ \text{mín}\Big\{r_j - \sum_{i=j+1}^{m} a_i w_i \ :\ 1 \leq j \leq m \ \wedge \ a_j = 1\Big\}\Big\}.$$

Es decir, r es el valor máximo que puede tomar el peso del próximo producto a agregar, pues si este lo supera implica que, o bien un producto se rompió con el peso de todos los otros productos por encima de el, o el tubo mismo se rompió. En caso de que eso suceda, podemos afirmar que cualquier solución que se extienda a partir de esta será una no válida, y por lo tanto, no vale la pena seguir explorando el árbol de backtracking. Esta es la motivación para la poda por factibilidad de este algoritmo.

La implementación de esta poda se puede observar en el Algoritmo 2. La idea de la implementación es ir actualizando el valor de r, la *resistencia mínima parcial* en cada llamado recursivo del algoritmo, dependiendo si se agrega o no un producto. Esto se puede observar en la línea 8. Para verificar la factibilidad de la solución parcial que estamos actualmente explorando, se realiza la comparación en la línea 3. En caso de ser una solución no factible, el llamado recursivo retorna $-\infty$, asegurándose de esta manera que siempre se opte por alguna otra rama.

Poda por optimalidad Sea $v \in \mathbb{R}^m$ la codificación de una solución parcial. Luego, podemos calcular el cardinal de la solución parcial como

$$c = \sum_{i=1}^{m} a_i.$$

Por otra lado, la cantidad de productos que quedan por considerar es $n-m$. Podemos concluir entonces que la cantidad máxima de productos que puede tener cualquier extensión de nuestra solución parcial es $c+(n-m)$. Supongamos entonces que max es el cardinal de alguna solución candidata ue ya hemos explorado, luego, si $c + (n - m) \leq max$, esta solución parcial, aunque se extienda de forma tal de incluir a todos los productos restantes, no podrá ser una solución óptima. Por lo tanto, no tiene sentido recorrer el subárbol producto de extender esta solución particular. Esta es la poda por optimalidad que consideramos para este algoritmo. La manera de implementar esta poda consiste en que la variable c (llamada *count* en el Algoritmo 2) se vaya incrementando a medida que agregamos un producto a la solución particular, y en caso de ser mayor a la variable max, que esta última se actualice con el valor de c, como se puede ver en la línea 7. Por último, la poda se realiza en la línea 6.

La complejidad del algoritmo en el peor caso es $O(2^n)$. Esto es así, porque en el peor escenario no se logra podar ninguna rama y por lo tanto se termina enumerando el árbol completo al igual que en el algoritmo de Fuerza Bruta. Además, se puede observar que el código introducido en las líneas 7, 3 y 6 solamente agrega un número constante de operaciones. Existe una familia de instancias para las cuales este algoritmo va a enumerar todo el árbol. Por ejemplo,

$$s_k = (1, n), \ldots, (1, n).$$

Algorithm 2 Optimizando Jambo-tubo por Backtracking.

```
max ← −1
function JT_BT(i, r, count, s_k)
    if r < 0 then return −∞
    if i = N then return 0
    restantes ← N − i
    if count + restantes ≤ max then return −∞
    if count > max then max ← count
    min ← mín(r − w_i, r_i)
    return máx(1 + JT_BT(i + 1, min, count + 1, s_k), JT_BT(i + 1, r, count, s_k))
```

Con $\mathcal{R} \geq n$. En este caso, cualquier subsucesión de $\{s_k\}$ representa una solución válida, dado que la suma de los pesos de todos los productos no es mayor a la resistencia del tubo ni de ningún producto. Por lo tanto no se puede aplicar en ningún momento la poda por factibilidad, resultando esto en una 2^n llamados recursivos. Notar que en la práctica el orden de los llamados recursivos determina si aplica o no una poda por optimalidad, dado que, se podría recorrer primero una rama que resulte en una solución óptima. En este caso, estamos considerando que los llamados recursivos se hacen simultáneamente y por ello, salvo una poda factibilidad, se está recorriendo el mismo nivel del árbol de backtracking para todo llamado recursivo. En cambio, el mejor caso se produce cuando el cardinal de la solución óptima es 0, esto es, no se puede agregar ningún producto sin que se supere la resistencia del tubo. En este caso, podemos observar que solo se enumeraran $O(n)$ nodos. Esto es gracias a la poda por factibilidad que corta en la siguiente iteración al ver que la resistencia del tubo es menor a 0. Un representante de esta familia de instancias es

$$s_k = (\mathcal{R}+1, \mathcal{R}), \ldots, (\mathcal{R}+1, \mathcal{R}).$$

4. Programación Dinámica

Los algoritmos de *Programación Dinámica* entran en juego cuando un problema recursivo tiene superposición de subproblemas. La idea es sencilla y consiste en evitar recalcular todo el subárbol correspondiente si ya fue hecho con anterioridad. En este caso, definimos la siguiente función recursiva que resuelve el problema:

$$f(i,m) = \begin{cases} -\infty & \text{si } m < 0, \\ 0 & \text{si } m \geq 0 \wedge i = n+1, \\ \text{máx}\{1 + f(i+1, m-w_i),\ f(i+1,m)\} & \text{si } m - w_i < r_i, \\ \text{máx}\{1 + f(i+1, r_i),\ f(i+1,m)\} & \text{caso contrario.} \end{cases} \tag{1}$$

Coloquialmente, podemos definir $f(i,m)$ como el máximo cardinal de una subsucesión $\{s_{k_r}\} \subseteq \{s_k\}$ que considere únicamente los últimos i productos de $\{s_k\}$, tal que la suma de los $w_j \in \{s_{k_r}\}$ es a lo sumo m. Más formalmente, queremos estudiar lo que pasa con el sufijo de la sucesión $\{s_k\}_{1 \leq k \leq n}$, que es exáctamente la sucesión $\{s_k\}_{i \leq k \leq n}$. Por simplicidad lo notaremos $\{s_k\}_i$. Veamos que la recursión es efectivamente lo que dice su definición coloquial.

Correctitud

(i) Si $m < 0$ entonces ningún subconjunto va a tener una suma menor o igual a m ya que todos los pesos w_j son enteros positivos, por lo tanto la respuesta $f(i,m) = -\infty$.

(ii) Si $i = n+1$ y $m \geq 0$, entonces $\{s_k\}_i$ es la sucesión vacía, y por lo tanto cualquier subsucesión tiene cardinal igual a 0, por lo tanto $f(i,m) = 0$.

(iii) En este caso, $i \leq n$ y $m \geq 0$, por lo tanto estamos buscando la subsucesión de $\{s_k\}_i$ de mayor cardinal que sume a lo sumo m. Dicha subsucesión tiene que, o bien tener al i-ésimo producto o no tenerlo. Si no lo tiene entonces tiene que ser a su vez una subsucesión de $\{s_k\}_{i+1}$ y no exceder m, por ende, debe hallarse de manera recursiva $f(i+1,m)$. Si tiene al i-ésimo elemento, entonces los w_j del resto de la solución deben sumar a lo sumo m', utilizando elementos de $\{s_k\}_{i+1}$, y debe ser la de máximo cardinal entre todas ellas. Esto es precisamente $f(i+1,m')$. A cada paso, el valor de m' se actualiza según el siguiente criterio:

a) si $m - w_i < r_i$ quiere decir que la resistencia más débil que no debemos quebrantar es la misma que en el paso anterior menos el peso del producto agregado, es decir que $m' = m - w_i$. Por lo tanto la solución es máx$\{1 + f(i+1, m - w_i),\ f(i+1, m)\}$.

b) Por el contrario, si $m - w_i \geq r_i$ entonces la resistencia más frágil es la del producto que acabamos de agregar. Por consiguiente, $m' = r_i$ y así la solución se configura como máx$\{1 + f(i+1, r_i),\ f(i+1, m)\}$.

Notar que al término de la izquierda se le suma 1 por haber seleccionado al i-ésimo elemento.

Memoización Notemos que la función recursiva (1) toma dos parámetros $i \in [1, \ldots, n]$ y $m \in [0, \ldots, R]$. Notar que los casos $i = n + 1$ o $m < 0$ son casos base y se pueden resolver de manera ad-hoc en tiempo constante. Por lo tanto, la cantidad de posibles *estados* con la que se puede llamar a la función, o combinación de parámetros, está determinada por la combinación de ellos. En este caso, hay $\Theta(n \cdot \mathcal{R})$ combinaciones posibles de parámetros. En este sentido, si agregamos una memoria que recuerde cuando un caso ya fue resuelto y su correspondiente resultado, podemos calcular una sola vez cada uno de ellos y asegurarnos no resolver más de $\Theta(n \cdot \mathcal{R})$ casos. El Algoritmo 3 muestra esta idea aplicada a la función (1). En la línea 7 se lleva a cabo el paso de memoización que solamente se ejecuta si el estado no había sido previamente computado.

Algorithm 3 Optimizando Jambo-tubo por Programación Dinámica

```
1: M_im ← ⊥ for i ∈ [1, n], m ∈ [0, R].
2: function JT_DP(i, m)
3:     if m < 0 then return −∞
4:     if M_im = ⊥ then
5:         if i = n + 1 then M_im ← 0 else
6:             min ← mín{m − w_i, r_i}
7:             M_im ← máx{1 + JT_DP(i + 1, min), JT_DP(i + 1, m)}
8:     return M_im
```

La complejidad del algoritmo entonces está determinada por la cantidad de estados que se resuelven y el costo de resolver cada uno de ellos. Como mencionamos previamente, a lo sumo se resuelven $O(n \cdot \mathcal{R})$ estados distintos, y como todas las líneas del Algoritmo 3 realizan operaciones constantes entonces cada estado se resuelve en $O(1)$. Como resultado, el algoritmo tiene complejidad $O(n \cdot \mathcal{R})$ en el peor caso. Es importante observar que el diccionario M se puede implementar como una matriz con acceso y escritura constante. Más aún, notar que su inicialización tiene costo $\Theta(n \cdot R)$, por lo tanto, el mejor y peor caso de nuestro algoritmo va a tener costo $\Theta(n \cdot \mathcal{R})$.

5. Experimentación

En esta sección se intentará evaluar y mostrar las fortalezas y debilidades de las diferentes técnicas algorítmicas presentadas en las secciones anteriores comparándolas contra distintas instancias. Para las ejecuciones de los experimentos se utilizó el lenguaje de programación *C++*, y estas se realizaron en una computadora con CPU AMD(R) FX(TM) 6100 @ 3.3 GHz y 8 GB de memoria RAM.

5.1. Métodos

Las configuraciones y métodos utilizados durante la experimentación son los siguientes:

- **FB**: Algoritmo 1 de Fuerza Bruta de la Sección 2.
- **BT**: Algoritmo 2 de Backtracking de la Sección 3.
- **BT-F**: Algoritmo 2 con excepción de la línea 6, es decir, solamente aplicando podas por factibilidad.
- **BT-O**: Similar al método BT-F pero solamente aplicando podas por optimalidad, o sea, descartando la línea 3 del Algoritmo 2.
- **DP**: Algoritmo 3 de Programación Dinámica de la Sección 4.

5.2. Instancias

Para evaluar los algoritmos en distintos escenarios es preciso definir familias de instancias conformadas con distintas características. Por ejemplo, el algoritmo de Backtracking como se menciona en la Sección 3 tiene familias que producen mejores y peores casos para el algoritmo. Los *datasets* definidos se enumeran a continuación.

- **fuerza-bruta**: Los tamaños de las instancias de este dataset varían entre 1 y 30. Para una instancia de n elementos, la resistencia del tubo es $\mathcal{R} = n$, y la sucesión de productos $\{s_k\}_{1 \leq k \leq n}$ está definida como

$$s_k = (1, \mathcal{R}) \quad k = 1, \ldots, n.$$

- **bt-mejor-caso**: Las instancias varían entre 1 y 30 elementos. Para una instancia de n elementos, la sucesión de productos $\{s_k\}_{1 \leq k \leq n}$ está definida como

$$s_k = (\mathcal{R} + 1, \mathcal{R}) \qquad k = 1, \ldots, n.$$

  Esta es una familia de instancias donde se exhibe el mejor caso para la técnica de Backtracking vista en la Sección 3. En este caso, la resistencia del tubo es $\mathcal{R} = 100$.
- **bt-peor-caso**: Las instancias varían entre 1 y 30 elementos. Para una instancia de n elementos con $\mathcal{R} \gg n$, la sucesión de productos $\{s_k\}_{1 \leq k \leq n}$ está definida como

$$s_k = (1, \mathcal{R}) \qquad k = 1, \ldots, n.$$

  Estas son algunas de las instancias donde la técnica de Backtracking, vista en la Sección 3, exhibe el peor caso de Backtracking.
- **sin-superposición**: Las instancias varían entre 1 y 20 elementos, $\mathcal{R} = 2^n - 1$ y la sucesión de productos $\{s_k\}_{1 \leq k \leq n}$ está dada por

$$s_k = (2^k, \mathcal{R}) \qquad k = 1, \ldots, n.$$

  Al construir las instancias de esta manera, garantizamos que nunca se llame al algoritmo de manera recursiva con los mismos parámetros más de una vez.
- **alta-superposición**: El tamaño de las instancias varía entre 1 y 20 elementos. Para una instancia de tamaño n, su resistencia está definida como $\mathcal{R} = \text{máx}(1, \lfloor \frac{n(n+1)}{4} \rfloor)$. Así mismo, la sucesión de productos $\{s_k\}_{1 \leq k \leq n}$ esta dada por

$$s_k = (k, \mathcal{R}) \qquad k = 1, \ldots, n.$$

  De esta manera nos aseguramos que para cada $0 \leq r \leq \mathcal{R}$, hay muchas subsucesiones $\{s_{k_r}\}_{1 \leq k_r \leq n_k}$ tales que

$$\sum_{s_{k_i} \in \{s_{k_r}\}} w_{k_i} = r.$$

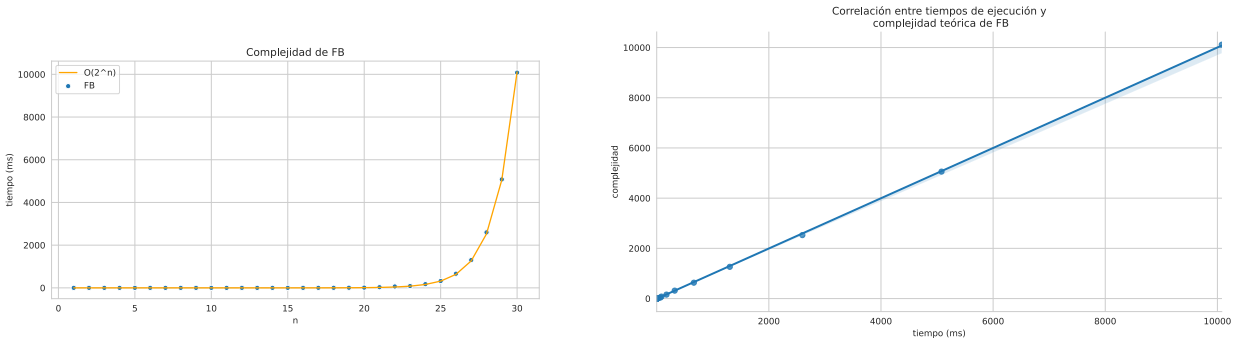
- **muchos-productos**: El tamaño de las instancias varían en el intervalo $[100, 1000]$ y su resistencia está dada por $\mathcal{R} = \lfloor \frac{n(n+1)}{4} \rfloor$, donde n es el tamaño de la instancia. Así mismo, la sucesión de productos $\{s_k\}_{1 \leq k \leq n}$ esta dada por

$$s_k = (k, \mathcal{R}) \qquad k = 1, \ldots, n.$$

5.3. Experimento: Complejidad de Fuerza Bruta

En este experimento se busca analizar los tiempos de ejecución del algoritmo de Fuerza Bruta y compararlos con la complejidad teórica del mismo vista en la Sección 2. Para este experimento utilizaremos el dataset **fuerza-bruta**. Esperamos que este experimento respalde empíricamente las lo señalado en la demostración de la complejidad teórica de este algoritmo en la Sección 2: principalmente que los tiempos de ejecución del algoritmo son exponenciales en n. Una observación que vale la pena mencionar es que, como vimos en la Sección 2, toda instancia exhibe el *peor caso* del algoritmo, y por lo tanto la construcción de las instancias utilizadas no son de particular interés.

La Figura 2 muestra los resultados del experimento. En la Figura 2a se puede observar que los tiempos de ejecución siguen un carácter exponencial en el tamaño de la instancia, n. Más sobre este punto, de la Figura 2b podemos ver que los tiempos de ejecución del algoritmo para todos los tamaños de instancias se correlacionan fuertemente (0.99996) con la complejidad teórica vista en la Sección 2, en particular que esta es del orden de $O(2^n)$. Estos resultados confirman nuestra hipótesis.



(a) Tiempos de ejecución FB

(b) Comparación con $O(2^n)$, Pearson: 0.99996

Figura 2: Análisis de complejidad del método FB.

5.4. Experimento: Complejidad de Backtracking

En esta experimentación vamos a contrastar las hipótesis de la Sección 3 con respecto a las familias de instancias de mejor y peor caso para el Algoritmo 2, y su respectiva complejidad. Para esto evaluamos el método **BT** con respecto los datasets **bt-mejor-caso** y **bt-peor-caso**.

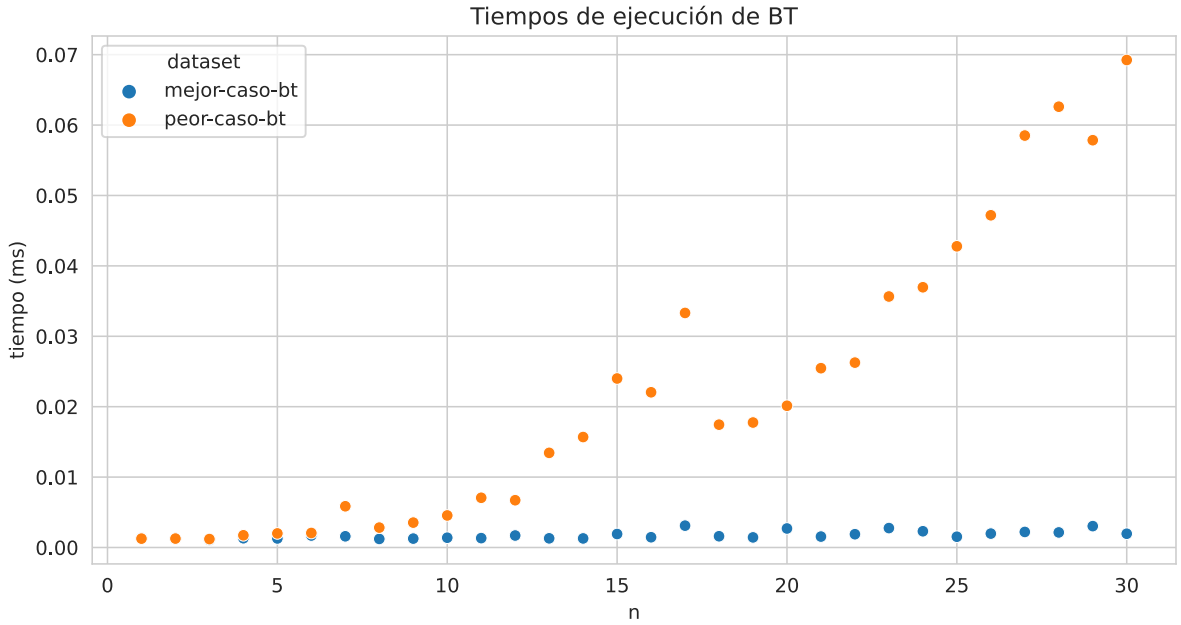
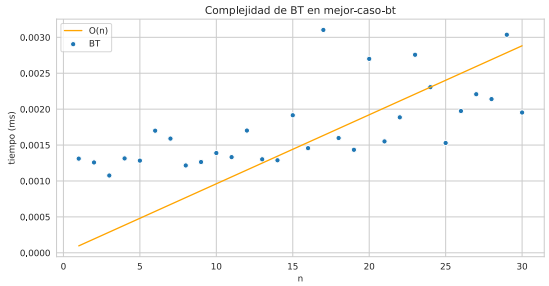


Figura 3: Comparación del tiempo de ejecución de mejor y peor caso

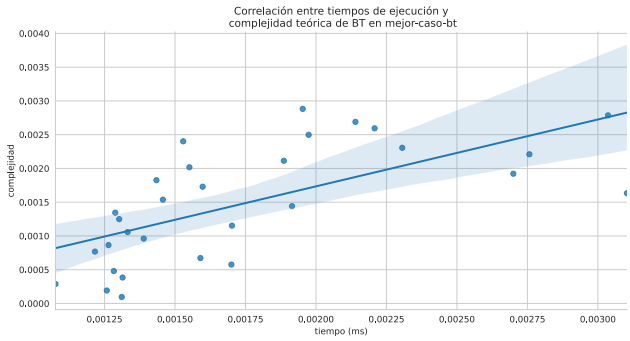
En primer lugar, la Figura 3 exhibe la diferencia de tiempos de ejecución entre los datasets **bt-peor-caso** y **bt-mejor-caso**.

Entrando más en detalle, las Figuras 4 y 5 muestran los gráficos de tiempo de ejecución y de correlación del método **BT** para cada dataset respectivamente.

Estudiando el dataset **bt-mejor-caso** podemos observar que existe una correlación positiva (0.7583) entre los tiempos de ejecución y la complejidad teórica (lineal en n), pero esta no es tan marcada como hubiésemos esperado. Así mismo, al analizar los resultados del dataset **bt-peor-caso**, también podemos ver que este exhibe una correlación positiva con la complejidad teórica (0.8052), pero tampoco es tan alta como hubiésemos esperado. En la Figura 5a podemos notar las diferencias entre los tiempos de ejecución y una curva exponencial en n. Se puede observar que para la mayoría de instancias, el tiempo de ejecución resultó mayor al esperado.



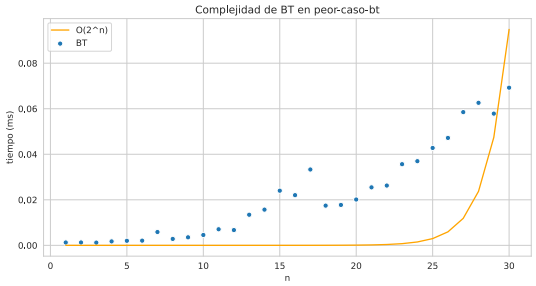
(a) Tiempos de ejecución mejor caso BT



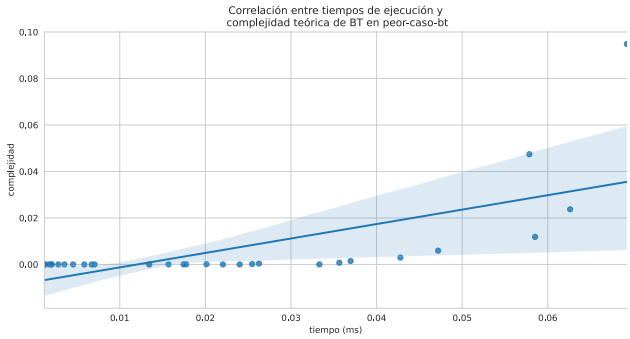
(b) Comparación con $O(n)$, Pearson: 0.7583

Figura 4: Análisis de complejidad del método BT para los dataset bt-mejor-caso

En ambos casos, los experimentos presentaron un menor grado de confianza con la complejidad teórica del que esperábamos. Esto pudo deberse a ruido presente a la hora de medir los tiempos de ejecución. Una posible manera de mitigar esto a futuro es realizar más mediciones para una determinada instancia, y remover *outliers* en las muestras.



(a) Tiempos de ejecución peor caso BT



(b) Comparación con $O(2^n)$, Pearson: 0.8052

Figura 5: Análisis de complejidad del método BT para el dataset bt-peor-caso.

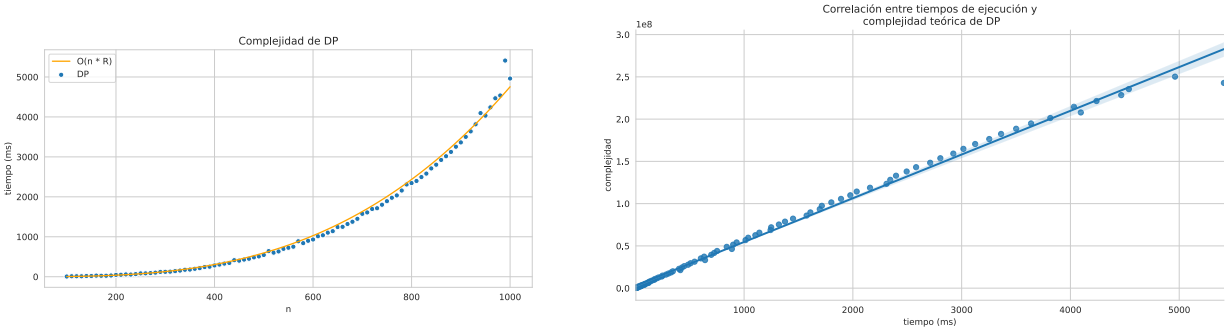
5.5. Experimento: Complejidad de Programación Dinámica

Para este experimento se quiere analizar la eficiencia del algoritmo de Programación Dinámica para un dataset determinado, y también ver su correlación con la cota teórica vista en la Sección 4. Para esto, mediremos los tiempos de ejecución del método **DP** sobre el dataset **muchos-productos**.

En la Figura 6a se exhibe el crecimiento del tiempo de ejecución del algoritmo comparado con una curva del orden de $O(n \cdot \mathcal{R})$. Podemos observar que la muestra adopta un comportamiento muy similar al de la curva. Más aún, en la Figura 6b puede verse la fuerte correlación (0.996) entre los tiempos de ejecución y la complejidad teórica. Estos resultados confirman nuestra hipótesis.

5.6. Experimento: Comparación Backtracking–Programación Dinámica

Para este experimento quisimos realizar una comparación entre dos técnicas algorítmicas que detallamos en secciones anteriores: Programación Dinámica y Backtracking. La motivación detrás de este experimento era poder tener un mejor entendimiento de, bajo qué instancias se comporta mejor una técnica por sobre la otra. Notemos que, no basta con saber la complejidad teórica de los algoritmos desarrollados con estas técnicas para determinar cuál tendrá menor tiempo de ejecución. Esto se debe a que ellas no dependen de los mismos factores, por ejemplo, por lo visto en las Secciones 3 y 2, la complejidad teórica del algoritmo de Programación Dinámica es del orden de $O(n \cdot \mathcal{R})$, mientras que la del algoritmo de Backtracking del orden de $O(2^n)$. Por lo tanto, dependiendo del valor de $\mathcal{R}$, podría llegar a pasar que la complejidad del algoritmo de Programación Dinámica supere el orden exponencial del algoritmo de Backtracking.

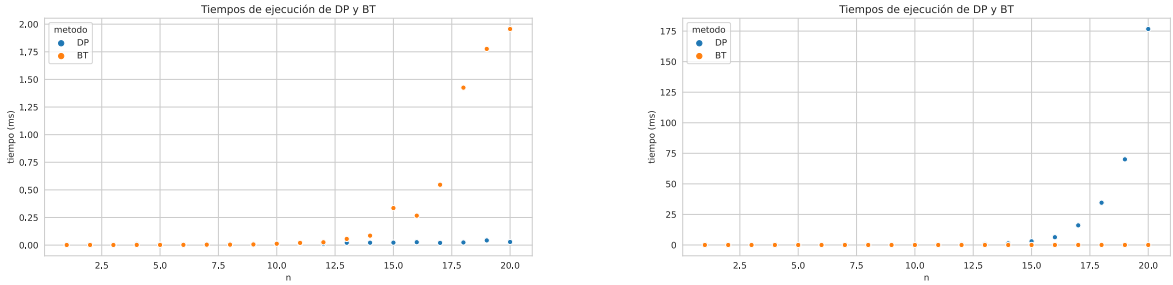


(a) Tiempos de ejecución PD.

(b) Comparación con $O(n \cdot \mathcal{R})$, Pearson: 0.996

Figura 6: Análisis del método DP sobre el dataset muchos-productos.

Nuestra hipótesis es que para instancias donde $\mathcal{R}$ y los valores que tome la resistencia mínima parcial varíe mucho dentro de un intervalo grande, el algoritmo de Backtracking presentará una ventaja frente al de Programación Dinámica debido al alto costo de mantenimiento de la estructura de memoización de este último y la efectividad de las podas del primero. Por el otro lado, para instancias donde $\mathcal{R}$ y los valores que tome la resistencia mínima parcial se mantengan acotados dentro de intervalo reducido, el beneficio de tener almacenados resultados a subproblemas se verá reflejado en los tiempos de ejecución.



(a) Dataset alta-superposicion.

(b) Dataset sin-superposicion.

Figura 7: Comparación de tiempos de ejecución entre DP y BT.

La Figura 7 muestra los tiempos de ejecución de los algoritmos de Backtracking y Programación Dinámica para los datasets **alta-superposición** y **sin-superposición**. Podemos observar que en ambos casos la hipótesis se confirma: en primer lugar, en el dataset **alta-superposición** el algoritmo de Programación Dinámica resulta más eficiente que el de Backtracking. Para la instancia con mayor productos de este dataset ($n = 20$), la resistencia resulta $\mathcal{R} = 105$, y por lo tanto vale que la resistencia mínima parcial, m, es $0 \leq m \leq \mathcal{R}$. Dado que los pesos de los productos de este dataset son todos los naturales hasta n, y $\mathcal{R}$ es el promedio de la suma de todos los elementos, existen muchas maneras de sumar a lo sumo $\mathcal{R}$. Por lo tanto, es esperable que muchos llamados recursivos coincidan, y esto es un factor determinante para determinar la efectividad de un algoritmo de Programación Dinámica. Por el otro lado, también se confirma la hipótesis con respecto al dataset **sin-superposición**. En este caso, se puede observar que el algoritmo de Backtracking es considerablemente más efectivo que el de Programación Dinámica. La resistencia $\mathcal{R}$ para la instancia de mayor tamaño de este dataset resulta $\mathcal{R} = 2^{20}$, y más aún el dataset es tal que nunca se aprovecha el almacenamiento de resultados de subproblemas. Notar que para $\mathcal{R}$ muy grande la complejidad espacial del algoritmo de Programación Dinámica empieza a ser un factor a considerar a la hora de determinar qué algoritmo usar. Por ejemplo, en la instancia de mayor tamaño del dataset **sin-superposición**, se requiere mantener una estructura de memoización de $20 \cdot 2^{20}$ entradas. Si suponemos que cada una de estas entradas ocupa 4 Bytes, el espacio que se necesita para la estructura de memoización es de 80 MB. Y si quisiéramos considerar 10 productos más que sigan el criterio de construcción del dataset **sin-superposición**, el espacio requerido pasaría a ser de 120 GB.

6. Conclusiones

En este trabajo se muestran tres algoritmos que usan técnicas distintas para resolver el problema de Jambo-Tubos. El algoritmo de Fuerza Bruta resulta poco eficiente para resolver este problema ya que al aumentar su tiempo de ejecución aumenta exponencialmente con la cantidad de elementos de $\{s_k\}$. Por este motivo, no resulta una elección recomendable para instancias que superen los 30 productos. Por otro lado, el algoritmo de Backtracking viene a mitigar las falencias del algoritmo anteriormente mencionado. Este introduce las podas por factibilidad y optimalidad, que en casos promedios mejoran las eficiencia del algoritmo considerablemente. En el mejor de los casos, el algoritmo presenta un tiempo de ejecución lineal en la cantidad de productos de la instancia. No obstante, el algoritmo no puede dar ninguna garantía en el análisis teórico, ya que para ciertas instancias el tiempo de ejecución del algoritmo es similar al de Fuerza Bruta. Por último, tenemos el algoritmo de Programación Dinámica que con el proceso de *memoización* busca evitar el cálculo de subproblemas que ya se han resuelto previamente. Este algoritmo permite considerar instancias de gran tamaño y resolverlas con una complejidad no exponencial en la cantidad de productos. Las desventajas de este algoritmo son que la complejidad depende de la resistencia $\mathcal{R}$ del Jambo-Tubo, y si esta es muy grande podría superar los tiempos de ejecución del algoritmo de Backtracking. La otra desventaja con este algoritmo es el mantenimiento y la complejidad espacial que implica la estructura de *memoización*.

Un área donde se podría profundizar a futuro es el impacto que el orden de cómputo de los llamados recursivos tiene frente al tiempo de ejecución del algoritmo de Backtracking. Así mismo, sería interesante investigar acerca de instancias, si existieren, donde el algoritmo de Fuerza Bruta presente mejoras en cuanto a su eficiencia por sobre el de Backtracking.