DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 1: Jumbo Tubos

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Belgorodsky, Martín	K310/I	wamozart@dc.uba.ar
Miguez, Tomás	94/19	tomasmiguez99@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

El problema a resolver a lo largo de este trabajo practico es el de los *Jambotubos*. Se tiene un tubo con una cierta resistencia R y una lista de productos, cada uno con una resistencia r_i y un peso w_i. Además, llamamos n al largo de la lista de productos. Queremos maximizar el *numero de productos* que empaquetamos en nuestro Jambotubo. Para que una lista de elementos empaquetados en un Jambotubo sea valida, el orden de los productos en dicha lista debe ser el mismo que en la entrada. También debe estar formada solo por elementos de la lista original, sin repetición. Además, la suma de los pesos de los elementos empaquetados en el Jambotubo no debe superar su resistencia, y la suma de los pesos de todos los elementos a partir de un cierto elemento, debe ser menor a la resistencia de dicho elemento.

Veamos algunos ejemplos con posibles soluciones:

Ejemplo 1:
R=50

r:	45	8	15	2	30
w:	10	20	30	10	15

Ejemplo 2:
R=100

r:	40	30	20	10	0
w:	10	10	10	10	10

Ejemplo 2:
R=5

r:	45	8	15	2	30
w:	10	20	30	10	15

Ejemplo 3:
R=100

r:	0	40	30	20	10
w:	10	10	10	10	10

Figura 1: En cada ejemplo en verde los productos empaquetados y en rojo los que no.

Para la resolución de este problema aplicaremos distintas técnicas algorítmicas, analizando la efectividad y eficiencia de cada una de ellas desde una perspectiva tanto teórica como empírica. En primera instancia aplicaremos *Fuerza Bruta* que consiste en enumerar todas las posibles soluciones y quedarnos con alguna de las factibles. Sobre este algoritmo recursivo, que genera un árbol de recesión, aplicamos distintas podas que permiten obviar casos redundantes, ya sea porque claramente seguir desarrollando esa rama no lleva a una solución (factibilidad) o porque no llevaría a una mejor solución que la que ya se tiene (optimalidad). Por ultimo, buscaremos aplicar *Programación Dinámica* para evitar hacer cálculos que resulten redundantes utilizando una estructura de memorización adecuada.

2. Observaciones Generales

Resistencia restante: Al describir el problema, dijimos que una solución sólo es válida si la suma de los pesos de los elementos de mi solución, a partir de un elemento i+1 cualquiera, es menor a la resistencia del elemento i. Este chequeo es caro, ya que al ver si podemos agregar un nuevo elemento, primero habría que recorrer todos los elementos que ya forman parte de la solución parcial. Pero observemos algo, suponga que definimos una resistencia limitante, que tiene cierto valor antes de ver un nuevo elemento, entonces para decidir si podemos o no agregar el nuevo elemento, vemos si el peso del nuevo elemento es menor que la resistencia restante, de serlo podemos agregar este elemento. Si decidimos agregarlo, la nueva resistencia restante pasa a ser el mínimo entre la resistencia del ultimo producto, y la anterior resistencia restante menos el peso del producto. Proponemos que este concepto, inicializando la resistencia restante como la resistencia del Jambotubo, es una condición equivalente a la planteada por el problema. De serlo, podríamos chequear qué productos se pueden agregar en tiempo constante, en lugar de $\mathcal{O}(n)$.
Veamos este concepto en el ejemplo 1. En el siguiente gráfico vemos la rama que lleva a la solución correcta. Cada columna es un nuevo producto que se va a agregar, con R Actual siendo la resistencia

restante antes de agregarlo, r Nuevo la resistencia del nuevo producto a agregar, y w Nuevo el peso del nuevo producto a agregar. Además, la ultima columna es de cuando se intento agregar el ultimo producto, cosa que no fue posible por su peso.

R Actual	50	40	15	5
r Nuevo	45	15	2	30
w Nuevo	10	30	10	15
Comparacion	(50-10) < 45?	(40-30) < 15?	(15-10) < 2?	

Figura 2: Rama que lleva a la solucion correcta del ejemplo 1, con resistencia restante.

Formalicemos esta idea, representamos nuestra solución como las listas ordenadas $\{r_i, \ldots, r_k\}$ y $\{w_i, \ldots, w_k\}$ que representan las resistencias y los pesos de los productos que forman nuestra solución respectivamente, con $0 \leq k \leq n$. Entonces, una de las condiciones para que nuestra solución sea valida es que, $\forall i \in \mathbb{N},\ 1 \leq i \leq k$, se verifica que $r_i \geq \sum_{j=i+1}^{k} w_j$. Proponemos que si definimos a la resistencia restante del paso i como $R_i := \text{mín}\{R_{i-1} - w_{i-1}, r_{i-1}\}$ entonces es equivalente a la condición del enunciado pedir que $\forall i \in \mathbb{N},\ 1 \leq i \leq k$ valga que $R_i \geq w_i$. Cabe aclarar que R_0 es la resistencia inicial del Jambotubo.

3. Fuerza Bruta

Un algoritmo de Fuerza Bruta enumera todo el conjunto de soluciones para el problema dado generando, en este caso, todas las subsecuencias posibles para la secuencia de datos dada. Generando así un conjunto de subsecuencias de S.

Pero este conjunto de subsecuencias no tiene todas las soluciones que resuelven el problema *Jumbotambos*. Para eso, presentamos el Algoritmo 1, el cual genera todas las soluciones posibles de forma recursiva. En el algoritmo, vamos decidiendo si incluir o no el siguiente elemento, y nos quedamos con la mejor de las dos ramas propuestas. En caso de llegar al caso base, allí verificamos si la rama actual es o no una solución factible, devolviendo así la cantidad de elementos agregados al tubo, es decir, incluídos, o 0 en caso de no ser una solución factible.

En la Figura 3 se ve un ejemplo del árbol de recursión para la instancia $S = \{1\frac{r45}{w10}, 2\frac{r8}{w20}, 3\frac{r15}{w30}\}$ y $R = 50$. Cada nodo intermedio del árbol representa una *solución parcial*, es decir, cuando aún no se tomaron todas las decisiones de qué elementos incluir, al llegar al pie del árbol, vemos las hojas, las cuales representan todas las subsecuencias posibles. La solución óptima $\{1\frac{r45}{w10}, 3\frac{r15}{w30}\}$ está marcada en verde, en celeste las otras soluciones factibles $\{\emptyset, \{2\frac{r8}{w20}\}, \{3\frac{r15}{w30}\}\}$, y en rojo el resto de subsecuencias que no son factibles. Notar que la solución al problema original es exactamente $recuFB(0, 0, R)$.

Algorithm 1 Algoritmo de Fuerza Bruta para Jumbotubos.

```
1: function recuFB(pos, sum, Rrem)
2:     if pos = n then
3:         if Rrem < 0 then return 0 else return sum
4:     return máx{recuFB(pos + 1, sum, R_rem),
5:     recuFB(pos + 1, sum + 1, mín{Rrem − W_pos, R_pos}}
```

La correctitud del algoritmo se basa en el hecho de que se generan todas las subsecuencias que se pueden hacer, dado que para cada elemento de S se crean dos ramas: una considerando el elemento en la subsecuencia y la otra excluyéndolo. Al haber generado todas las subsecuencias, se descartan las que no son factibles para resolver el problema, y se encuentra la óptima (de existir).

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 *pos* es incrementado en 1 hasta llegar a n.

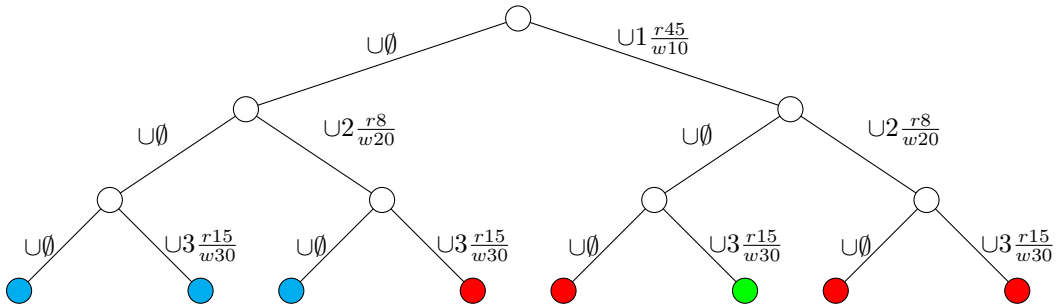


Figura 3: Ejemplo de ejecución del Algoritmo 1 para $S = \{1\frac{r45}{w10}, 2\frac{r8}{w20}, 3\frac{r15}{w30}\}$ y $R = 50$.
En verde la solución óptima $\{1\frac{r45}{w10}, 3\frac{r15}{w30}\}$, en celeste las otras soluciones factibles, y en rojo el resto de subsecuencias que no son factibles.

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 suma, mínimos, máximos 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 busca la misma cantidad de subsecuencias. Es decir, el cardinal de instancias buscadas en el mejor caso es igual al de peor caso, para un mismo n.

4. Backtracking

Los algoritmos de *Backtracking* siguen una idea similar a *Fuerza Bruta* pero con algunas consideraciones especiales. De la misma forma, se generan, en este caso, todas las subsecuencias que se pueden hacer con la secuencia dada, formando un *árbol de backtracking*. Pero en este caso, el Algoritmo puede generar *podas*, las cuales ayudan a descartar ramas y así reducir los tiempos de ejecución, esta exclusión de ramas se realiza en base a: *factibilidad* y *optimalidad*.

Poda por factibilidad En este problema planteado, una poda por factibilidad es la siguiente. Sea S' una subsecuencia parcial representada en un nodo intermedio n_k con $k = \#S'$. Vemos que para que sea una subsecuencia válida, se tiene que cumplir que $Rrem \geq 0$ todo el tiempo, que es la resistencia restante actual. Esto lo vemos en la línea 8 del Algoritmo 2.

Poda por optimalidad Una vez que tenemos una solución que podría resolver el problema, buscaríamos cuál es la mejor solución que resuelve el problema. Para esto, supongamos que se está en un nodo intermedio $n_{sum'}$ con $sum' = \#S'$ que representa a una subsecuencia parcial S'. Vemos que si $sum' + (n - pos') < S$ con pos' la posición que está recorriendo en ese momento de la secuencia inicial, entonces esa rama no tiene ninguna de probabilidad de agregar tantos elementos necesarios como para superar a la mejor solución de ese momento (S). Esta poda evita generar subsecuencias que no van a generar soluciones superadoras a la que hay hasta el nodo $n_{sum'}$. En el Algoritmo 2 se actualiza la variable global S cada vez que se halla una subsecuencia factible en la línea 5, y se evalúa la condición de la poda, antes mencionada, en la línea 7.

La complejidad del algoritmo en el peor caso es $O(2^n)$, al igual que en el de *Fuerza Bruta*. Esto se debe a que en el peor escenario no se logra podar ninguna rama por factibilidad ni por optimalidad, entonces se terminan buscando todas las subsecuencias posibles, completando el árbol. Las líneas 5, 7 y 8 solo suman operaciones de tiempo constante, por lo que no altera el orden de la complejidad del Algoritmo 2. Este peor caso sucede en una familia de instancias para las cuales este algoritmo va a enumerar todo el árbol, que son aquellas en las que todos los elementos de la secuencia comprenden la solución final. En caso contrario, el mejor caso sucede cuando podemos podar el árbol más rápido, particularmente, todos los elementos deberían estar pasados del límite de la resistencia del tubo, y no permitir que entren ninguno. Así, la poda por

Algorithm 2 Algoritmo de Backtracking para Jumbotubos.

```
1: S ← 0
2: function recuBT(pos, sum, Rrem)
3:     if pos = n then
4:         if Rrem < 0 then return 0
5:         S ← max{S, sum}
6:         return sum
7:     if S ≥ sum + (n − pos) then return 0
8:     if Rrem < 0 then return 0
9:     return máx{recuFB(pos + 1, sum, R_rem),
10:    recuFB(pos + 1, sum + 1, mín{Rrem − W_pos, R_pos}}
```

optimalidad garantizaría que ningún nodo se va a ramificar, sino que solamente se va a generar una solución que es la subsecuencia vacía, y por lo tanto, la poda por factibilidad garantizaría que no se enumeran más de $O(n)$ nodos.

5. 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:

$$f(i,r) = \begin{cases} 0 & \text{si } r \leq 0, \\ 0 & \text{si } i = n+1, \\ \text{máx}\{f(i+1,r), & \\ 1 + f(i+1, \text{mín}\{r - w_i, r_i\})\} & \text{caso contrario.} \end{cases} \tag{1}$$

Sea $P^i = \{p_i, \ldots, p_n\}$ el conjunto de productos con resistencias $\{r_i, \ldots, r_n\}$ y pesos $\{w_i, \ldots, w_n\}$. Coloquialmente, podemos definir $f(i,r)$: "máximo cardinal de un subconjunto de P^i que sea un empaquetamiento válido para un Jumbotubo de resistencia restante r". Claramente, $f(1,R)$ es el "maximo cardinal de un subconjunto de productos que sea un empaquetamiento valido de un jumboTubo de resistencia inicial R" lo cuál es exactamente la solución de nuestro problema. Veamos que la recesión es efectivamente lo que dice su definición coloquial.

Correctitud

(i) Si $r \leq 0$ entonces claramente el único subconjunto valido es el vacío, ya que todos los objetos tienen peso, y como el cardinal del conjunto vacío es 0, este caso es correcto.

(ii) Si $i = n + 1$ entonces quiere decir que buscamos subconjuntos de $\emptyset$ que sean empaquetamientos validos, el único subconjunto del conjunto vacío es el conjunto vacio, por lo cual 0 es el valor al que debe evaluar.

(iii) En este caso, $i \leq n$ y $r > 0$ entonces estamos efectivamente buscando un subconjunto de P^i que sea empaquetamiento valido. De existir dicho subconjunto, tiene que o bien tener al i-ésimo producto o no tenerlo. Si no lo tiene, entonces tiene que ser a su vez un subconjunto de P^{i+1} y ser empaquetamiento valido con resistencia restante $R_{i+1} = r$ ya que no agregamos un nuevo producto, por lo tanto, debe encontrarse de manera recursiva $f(i+1,r)$. Si tiene al i-ésimo producto, entonces la resistencia restante para el siguiente paso recursivo es $R_{i+1} := \text{mín}\{R_i - w_i, r_i\}$, utilizando elementos de P^{i+1} y debe ser la de máximo cardinal entre todas ellas. Esto es precisamente $f(i+1, \text{mín}\{r - w_i, r_i\})\}$. Por lo tanto, la mejor solución es

$f(i, r) = \text{máx}\{f(i+1, r), 1 + f(i+1, \text{mín}\{r - w_i, r_i\})\}$. Notar que al término de la derecha se le suma 1 por haber seleccionado al i-ésimo producto.

Memorización Primero hagamos una aclaración sobre r. Las sucesivas llamadas de f tienen valores decrecientes de r, ya que si no se utiliza al i-ésimo producto en la solución, entonces se vuelve a llamar con el mismo valor de r a la función, pero si utilizamos a dicho producto, entonces el nuevo valor de r será $\text{mín}\{r - w_i, r_i\}$, que como los pesos son positivos, este valor es menor a r. De todo esto, podemos concluir que si se invoca a f con $r = R$, entonces las llamadas recursivas de f se hacen con valores de $r \in [1, R]$, excepto en los casos base.
Además, si invocamos a f con $i = 1$, vemos que en todas las llamadas recursivas $i \in [1, n]$, excepto en los casos base que igualmente no es necesario memorizar.
De esto, se ve que hay un limite superior al numero de estados diferentes que no son triviales de calcular, y que dicho limite es $n * R$. Por otro lado, si todos los productos entran en nuestro Jumbotubo, entonces el numero de llamadas recursivas de f que habría que hacer es de 2^n, ya que nunca tenemos que la resistencia restante sea menor o igual a 0, excepto quizás en el ultimo producto.
A continuación, se muestra el pseudocódigo de una posible implementación de nuestro algoritmo, con una matriz como estructura de memorización.

Algorithm 3 Algoritmo de Programación Dinámica para Jumbotubos.

```
1: M_ir ← ⊥ for i ∈ [1, n], w ∈ [1, R].
2: function DP(i, r)
3:     if r ≤ 0 then return 0
4:     if i = n + 1 then return 0
5:     if M_ir = ⊥ then M_ir ← máx{DP(i + 1, r), 1 + DP(i + 1, mín{r − w_i, r_i})}
6:     return M_ir
```

De todo esto, podemos ver que la complejidad temporal de nuestro algoritmo pertenece a $\mathcal{O}(2^n)$ sin memorización, pero pertenece a $\mathcal{O}(n * R)$ al memorizar, ya que ese es el numero maximo de estados diferentes que debemos computar, y cada estado se resuelve en tiempo constante, como se ve en el pseudocodigo. Por lo que si $2^n >> n * R$, y se dispone del espacio necesario, entonces convendría memorizar.

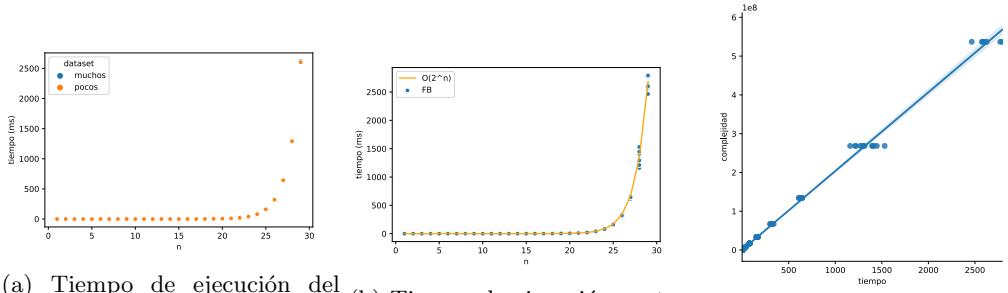
6. Experimentación

En esta sección se presenta los experimentos computacionales realizados para evaluar los distintos métodos presentados en las secciones anteriores. Las ejecuciones fueron realizadas en una workstation con CPU Intel Core i7 @ 2.8 GHz y 16 GB de memoria RAM, y utilizando el lenguaje de programación *C++*.

6.1. Métodos

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

- **FB**: Algoritmo 1 de Fuerza Bruta de la Sección 3.
- **BT**: Algoritmo 2 de Backtracking de la Sección 4.
- **BT-F**: Algoritmo 2 con excepción de la línea 7, 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 8 del Algoritmo 2.
- **DP**: Algoritmo 3 de Programación Dinámica de la Sección 5.



(a) Tiempo de ejecución del método *recuFB* sobre "pocosz "muchos".

(b) Tiempo de ejecución contra complejidad esperada.

(c) Correlación entre el tiempo de ejecución y la complejidad esperada.

Figura 4: Análisis de complejidad del método *recuFB*.

6.2. Instancias

Para ayudar a verificar empíricamente el análisis del mejor y peor caso, además de permitir hacer una comparación practica entre los distintos algoritmos, creamos una serie de datasets de prueba. Los mismos se encuentran descritos a continuación.

- **muchos-productos**: En este dataset tenemos casos de prueba en el que el numero de productos que entran en nuestro Jumbotubo es máximo, para lograr esto creamos todos los productos con peso 0.
- **pocos-producto**: Para distintas cantidades de productos, todos los productos pesan mas que la resistencia del Jumbotubo, por lo que no debería entrar ninguno.
- **uniforme**: Para ese dataset, la única restricción es que los pesos y resistencias tienen que estar entre 1 y R.

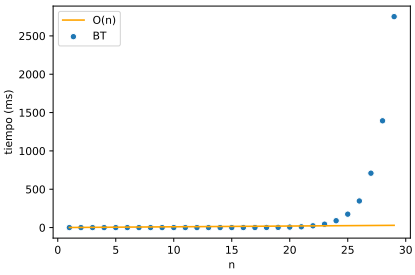
6.3. Experimento 1: Complejidad de Fuerza Bruta

En este experimento se analiza la desempeño del método *recuFB* en distintas instancias. El análisis de complejidad realizado en la Sección 3 indica que el tiempo de ejecución para el mejor y peor caso es idéntico y perteneces a $O(2^n)$. Para contrastar empíricamente estas afirmaciones se evalúa *recuFB* utilizando los datasets de "muchosz de "pocosz graficamos los tiempos de ejecución en función de n.

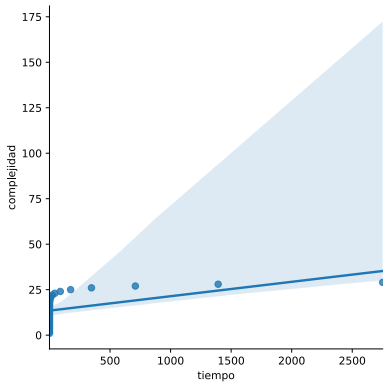
La Figura 4 presenta los resultados del experimento, donde se puede apreciar que ambas curvas están solapadas en todas las instancias (la azul se ve tapada por la naranja). En este gráfico se puede apreciar que los tiempos de ejecución no se alteran si damos una instancia con solución vacía o una solución que comprenda la secuencia entera de entrada, ambas siguen la misma curva de crecimiento exponencial.

Adicionalmente, tomamos la ejecución sobre todos los datasets y evaluamos su correlación con la complejidad $O(2^n)$, la cual estudiamos en la Sección 3. En la Figura **??** se ilustra el tiempo de ejecución de *recuFB* similar al de una función exponencial de $O(2^n)$. Y por último, en la Figura **??** aparece un *gráfico de correlación*, en el cual se compara el tiempo de ejecución real transformado en una función contra el tiempo esperado, en caso de que se comporten de igual manera, debería verse un gráfico de forma lineal, tal cual como se ve.

El tiempo de ejecución se ve que va a la par de una curva exponencial 2^n, la correlación respeta una recta, y el índice de correlación de Pearson de ambas variables es $r \approx 0{,}99829$. Así, concluímos en que la hipótesis planteada corresponde a como realmente se comporta el algoritmo.

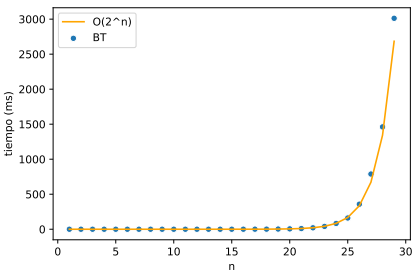


(a) Tiempo de ejecución vs Complejidad esperada.

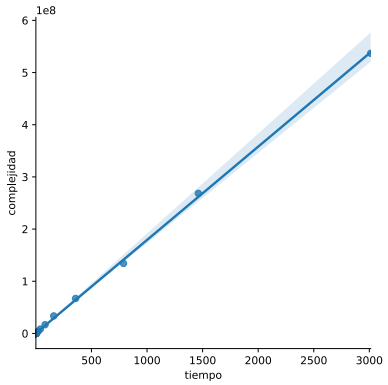


(b) Gráfico de correlación entre el tiempo de ejecución y la complejidad esperada.

Figura 5: Análisis de complejidad del método *recuBT* para el dataset "pocos".



(a) Tiempo de ejecución vs Complejidad esperada.



(b) Gráfico de correlación entre el tiempo de ejecución y la complejidad esperada.

Figura 6: Análisis de complejidad del método *recuBT* para el dataset "muchos".

6.4. Experimento 2: Complejidad de Backtracking

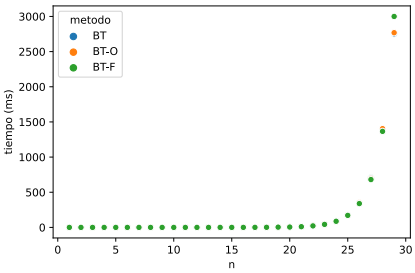
En esta experimentación vamos a contrastar las hipótesis de la Sección 4 con respecto a las familias de instancias de mejor y peor caso para el Algoritmo 2, y su respectiva complejidad.

En este caso evaluaremos los datasets "pocosz "muchos", que reflejarían el mejor y peor caso respectivamente, con el método *recuBT*.

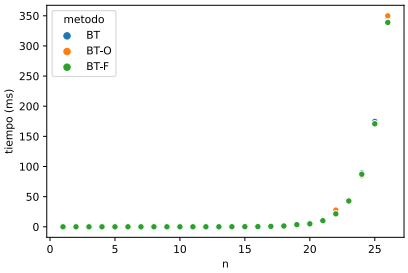
Las Figuras 5 y 6 muestran los gráficos de tiempo de ejecución de *recuBT* y de correlación para cada dataset respectivamente. Vemos que para el para el mejor caso, no pudimos comprobar la hipótesis planteada, para la cual las podas debían hacer que la complejidad se convierta en lineal; sino que en su defecto, observamos que continúa siendo de forma exponencial como su peor caso, que veremos a continuación.

Las podas planteadas no verificamos que sean efectivas para reducir la complejidad para los casos estudiados. Para el mejor caso, el índice de correlación de Pearson es $r \approx 0{,}53031$ lo que nos indica que efectivamente no sigue la complejidad lineal, y probablemente no hayamos estudiado un mejor caso efectivo.

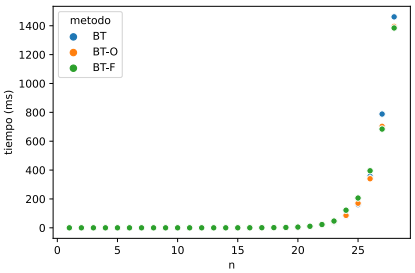
Por otra lado, las instancias de peor caso se ve el no se ve este comportamiento, y los tiempos de ejecución se presentan más ajustados a la curva de complejidad exponencial. Para estas instancias el índice de correlación de Pearson es de $r \approx 0{,}99980$ contra una función exponencial 2^n, comprobando así que la hipotesis planteada anteriormente se cumple, ya que se generaron todas



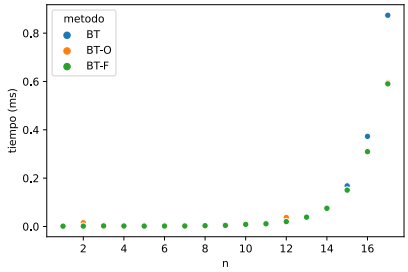
(a) Efectividad de las podas para "pocos".



(b) Efectividad de las podas con zoom para "pocos".



(c) Efectividad de las podas para "muchos".



(d) Efectividad de las podas con zoom para "muchos".

Figura 7: Comparación de efectividad en las podas.

las subsecuencias posibles, es decir, el árbol completo.

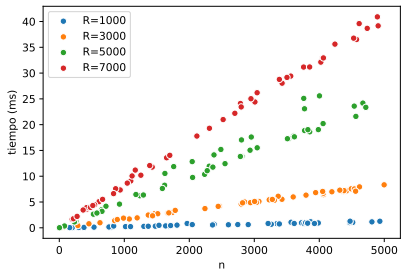
6.5. Experimento 3: Efectividad de las podas

En este experimente, verificamos qué podas son más efectivas que otras. Para eso verificaremos las podas por factbilidad, optimalidad y ambas juntas (BT-F, BT-O y BT respectivamente). Las hipótesis indican que las podas deben reducir la complejidad para un grupo grande de instancias, volviéndolos más eficientes que un algoritmo de *Fuerza Bruta*. Esto lo vemos nuevamente con los datasets "pocosz "muchos", deberíamos ver que los algoritmos son más eficientes con "pocos".

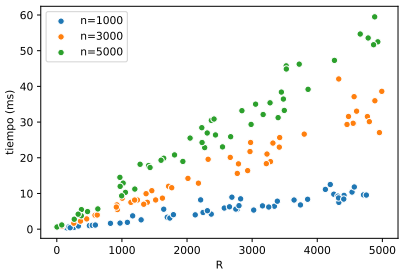
Verificamos que al igual que en el Experimento 6.4, no podemos concluir con que se cumplen las hipótesis, ya que en todos los casos, por más podas que apliquemos, la complejidad no baja, y siempre mantiene una forma exponencial de base 2 y exponente n.

6.6. Experimento 4: Complejidad de Programación Dinámica

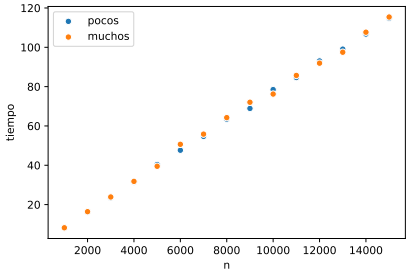
En primera instancia nos preguntamos si se existen peores y mejores casos para un n y R dado al resolver con PD. Para esto, analizamos el código. En este observamos que el numero de llamadas recursivas estaba estrechamente ligado al numero de productos que entran en nuestra mejor solución (a mas productos, mas llamadas recursivas). Y el mejor caso sería cuando ningún producto entra en nuestro tubo, en el cual solo habría que recorrer cada producto una vez, por una complejidad de $\mathcal{O}(n)$. Para verificar nuestra hipótesis, corrimos nuestro algoritmo sobre los datasets muchos-productos y pocos-productos, el resultado de dicho experimento se puede observar en el gráfico 8c, el cual deja en evidencia que nuestra hipótesis fue incorrecta. Esto se debe a que, aunque el algoritmo en el mejor caso se resuelve en $\mathcal{O}(n)$, la declaración e inicialización de la matriz de memorización sigue tomando tiempo $\mathcal{P}(n * R)$.
A continuación se analiza la eficiencia del algoritmo de Programación Dinámica en la práctica y



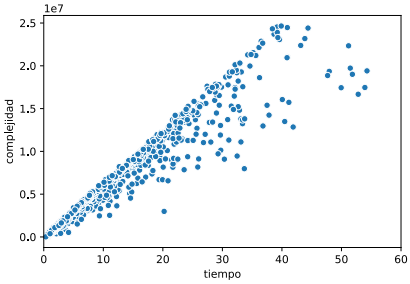
(a) Tiempo de ejecución en función de n.



(b) Tiempo de ejecución en función de R.



(c) Comparación del tiempo de ejecución en relación al número de elementos en la solución.



(d) Correlación entre el tiempo de ejecución y la cota de complejidad temporal.

Figura 8: Resultados computacionales para el método DP sobre el dataset uniforme.

su correlación con la cota teórica calculada en la Sección 5. Para esto, fijamos en distintos valores a la variable R y graficamos el tiempo de ejecución contra n, en el gráfico 8a; y viceversa en el gráfico 8b. Todas las curvas de dichos gráficos muestran un comportamiento lineal, dentro de un margen debido a la variación entre el mejor y peor caso.
Finalmente, graficamos los valores obtenidos de tiempo de ejecución contra la complejidad teórica esperada de $\mathcal{O}(n * R)$ y vemos que la correlación es elevada, con un coeficiente de Pearson por arriba del 0,95.

6.7. Experimento 5: Backtracking vs Programación Dinámica

A la hora de elegir cual algoritmo es mejor, observemos que la complejidad de PD en el peor caso es de $\mathcal{O}(n * R)$, osea que depende de n y de R, mientras que la de BT es $\mathcal{O}(2^n)$, que sólo depende de n, por lo que para valores elevados de R, convendría utilizar BT, mientras que cuando n crece a una velocidad similar a n, conviene utilizar PD.

7. Conclusiones

En este trabajo se presentan tres algoritmos que usan técnicas distintas para resolver el problema de empaquetado de Jumbotubos. El algoritmo de Fuerza Bruta es poco eficiente para resolver este problema ya que al aumentar el número de productos rápidamente crece su tiempo de ejecución a tiempos inmanejables. Una mejora a este algoritmo es el de Backtracking con sus podas que demuestran ser de utilidad en todas las instancias. Por último, el algoritmo de Programación Dinámica es el más robusto frente al crecimiento de la variable n, aunque es más sensible a el tamaño de R lo que hace que ante valores de R muy grandes no sea la mejor elección.