DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP1 - Optimizando Jambo-tubos

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Bukovits, Nicolás Axel	546/14	nicko_buk@hotmail.com
Chen, Alejandro Antonio	507/17	chenalejandro@outlook.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 quiere construir un robot con el fin de optimizar las operaciones en sus locales. Uno de ellos es el servicio de empacado de productos en los denominados jambos-tubos, donde en cada tubo se apilan los distintos productos, uno arriba del otro, es decir, un producto no puede estar ubicado al lado de otro, solamente arriba o abajo.

Como es de conocimiento general, hay ciertos productos que si tienen cierto peso por encima se aplastan, lo cual es una situación indeseable. Por lo cual, vamos a tener que resolver este problema decidiendo qué productos apilar para no aplastar los otros que están abajo. Entonces, sabiendo el orden de la llegada de los productos y sus respectivos pesos y resistencias, es decir, cuánto peso pueden tener arriba sin ser aplastados, decidir cuáles de los productos poner en el tubo. Observar que el tubo tiene también su resistencia propia, o sea que tiene un límite de peso total que se puede cargar.

Vamos a llamar R a la resistencia del tubo, n a la cantidad de productos, S a la secuencia ordenada de n productos, w_i al peso de cada producto y r_i a sus respectivas resistencias asociadas. El problema consiste en determinar la máxima cantidad de productos que pueden apilarse en el tubo, sin que ninguno esté aplastado. Notar que se debe respetar el orden dado y que todos los valores mencionados son enteros no negativos. Por ejemplo, dado un tubo con $R = 50$, $n = 5$, $w = [10, 20, 30, 10, 15]$ y $r = [45, 8, 15, 2, 30]$, la solución óptima es 3, y consiste en tomar los elementos 1, 3 y 4. Observar que tomando los elementos 1, 3 y 5 no es una solución factible ya que la suma de sus pesos es $55 > R$. Otro ejemplo, en donde no hay una solución, puede ser con $R = 50$, $n = 3$, $w = [55, 65, 60]$ y $r = [20, 10, 5]$. En este caso no se puede agregar ningún producto, porque se aplastan y porque todos pesan más que la resistencia del tubo, por lo que la respuesta al problema es 0.

La finalidad de este trabajo es resolver el problema presentado de los jambo-tubos, utilizando tres técnicas de programación distintas. La primera consiste en un algoritmo de fuerza bruta recursivo, que enumera todas las posibles soluciones quedándose con la mejor posible de las factibles. La segunda técnica consiste en un algoritmo de backtracking implementado con podas para reducir la cantidad de nodos del árbol con el objetivo de que sea más efeciente. Y finalmente se presentará un algoritmo recursivo de programación dinámica, utilizando memoización para evitar repetir los cómputos de los subproblemas. Posteriormente se realizará una experimentación comparando estos tres algoritmos con distintos tipos de instancias para ver como se comportan en distintas situaciones. Se cuenta para todos los algoritmos con un vector de longitud n, denominado *productos* que tiene en cada posición una estructura que representa a un producto con su peso y resistencia. Asi, por ejemplo el peso y la resistencia del producto i se accede con $productos[i].peso$ y $productos[i].resistencia$ respectivamente.

2. Fuerza Bruta

La técnica algorítmica de Fuerza Bruta consiste en enumerar todo el conjunto de soluciones en búsqueda de aquellas factibles u óptimas según si el problema es de decisión u optimización. En este caso, se trata de un problema donde se busca el número máximo de productos que se pueden apilar, dadas unas restricciones. El conjunto de soluciones está compuesto entonces por todos los subconjuntos de S, es decir, es el *conjunto de partes de* S que se escribe $\mathcal{P}(S)$.

La idea del Algoritmo 1 para resolver el problema del jambo-tubo es ir generando las soluciones de manera recursiva decidiendo en cada paso si un producto de S es agregado o no y quedándose con la mejor solución (el máximo) de alguna de las dos ramas. Cuando se llega a una solución (que es cuando se analizan todos los productos, hayan o no hayan sido agregados), se verifica que la solución es válida, es decir, cumple con los requisitos del problema, en este caso que ningún producto se aplaste y que la suma de los pesos w_i no supere la resistencia del tubo R. En caso

de sér válida se devuelve la cantidad de productos apilados. En caso de no ser válida se devuelve 0. Para este algoritmo se usa una variable global denominada $solucionParcial$, que es un vector de valores booleanos de longitud n inicializado completamente en $false$, que se utiliza para saber si un producto fue o no agregado. Cuando es agregado el producto i se almacena en la posición $solucionParcial[i]$ el valor $true$, caso contrario el valor $false$.

En la Figura 1 se ve un ejemplo del árbol de recursión para la instancia $n = 3$. 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 es $\{w_1, w_2\}$. El resto de las hojas o tienen menor o igual que un solo elemento o no es una solución factible.

Algorithm 1 Algoritmo de Fuerza Bruta para Jambo-tubos.

```
1: function FB(i, k, p)
2:     if i = n then
3:         if esSolucionValida(p) then return k else return 0
4:     solucionParcial[i] ← false
5:     sinAgregar ← FB(i + 1, k, p)
6:     solucionParcial[i] ← true
7:     agregado ← FB(i + 1, k + 1, p + productos[i].peso)
8:     return máx{sinAgregar, agregado}.
```

La correctitud del algoritmo se debe a que se generan todas las posibles soluciones, dado que para cada elemento de S se crean dos ramas una agregandólo al producto y la otra no agregándolo. Cuando no se agrega al producto simplemente se pasa al siguiente. Cuando es agregado el producto, se incrementa en uno el valor de k que representa la cantidad de productos agregados, y se agrega también el peso del producto agregado a p, que contiene el peso acumulado de los productos agregados. Al haber generado todas las posibles soluciones, debe encontrarse la óptima (de existir), caso contrario el valor del óptimo será 0, ya que no es posible apilar ningun próducto. La óptima va a ser encontrada mediante la llamada a la funcion max que calcula el valor máximo entre las dos ramas consideradas. El resultado al problema se obtiene llamando a la función $FB(0, 0, 0)$.

La complejidad del Algoritmo 1 para el peor caso es $O(n * 2^n)$. Esto se debe a que 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. La solución de cada llamado recursivo toma tiempo constante dado que se realizan solo sumas, asignaciones a vectores y comparaciones, excepto el caso cuando $i = n$. En este caso, que correspondería a las hojas del árbol, cuando se terminaron de analizar todos los productos, se llama a una función cuyo costo es de $O(n)$. El pseudocódigo de la función está en el algoritmo 2.

En dicho algoritmo se puede deducir que la complejidad de la función que determina si los productos resisten el peso encima ($todosLosProductosResistenPeso$) es $O(n)$ en el peor caso, ya que se realizan todas operaciones en tiempo constante, y sólo hay un ciclo que en el peor caso itera n veces (cuando es llamada con el parámetro n). Notar que el ciclo corta ni bien encuentra un producto que no resiste el peso encima de él. Para saber cuál es el peso encima de cada producto, al peso total recibido se le va restando el peso de cada producto y verificando que el peso restante no sea mayor a la resistencia del producto para el producto i.

Con lo expuesto anteriormente, también se puede pensar que si bien la complejidad del algoritmo de fuerza bruta es $O(n * 2^n)$, el algoritmo va a presentar una mejor performance en instancias en donde pocos productos puedan ser apilados (con el mejor caso siendo cuando ningún producto se puede apilar) y se va a comportar de una peor manera en tiempos de ejecución en instancias en donde varios productos se pueden apilar (con el peor caso siendo que todos los productos se puedan apilar). La causa de esta hipótesis es que el ciclo que verifica que los productos se pueden o no apilar, corta ni bien un producto no se puede apilar, es decir, no itera siempre hasta el final, por lo que si ningún producto se puede apilar, no itera ni una sola vez. Caso contrario, si todos se pueden apilar iterará hasta n. Obviamente para las soluciones candidatas en donde no se agre-

Algorithm 2 funciones para verificar si una solución encontrada es válida

```
1:  function esSolucionValida(p)
2:      return todosLosProductosResistenPeso(n, p) and p <= R.
3:  function todosLosProductosResistenPeso(i, p)
4:      j = 0
5:      pesoEncima = p
6:      while j < i and resistePeso(j, pesoEncima) do        ▷ corta si producto j se aplasta
7:          if solucionParcial(j) then
8:              pesoEncima ← pesoEncima − productos[j].peso
9:          j++
10:     return j = i
11: function resistePeso(i, p)
12:     if solucionParcial(i) then
13:         return p − productos[i].peso <= productos[i].resistencia
14:     return true
```

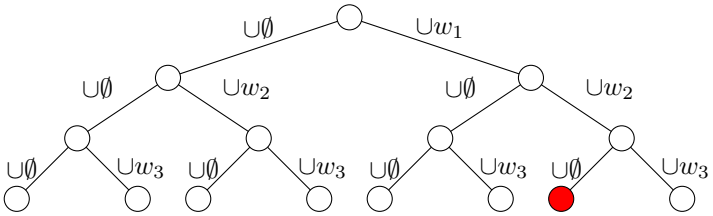


Figura 1: Ejemplo de ejecución del Algoritmo 1 para $n = 3$, $w = \{10, 10, 20\}$, $r = \{15, 18, 10\}$ $R = 50$. En rojo la solución óptima $\{w_1, w_2\}$.

gan productos, con el caso extremo de la solucion en donde no se considera agregar ninguno, la iteracion se realizará hasta n ya que la función de *resistePeso* devuelve verdadero si el producto no fue agregado. Esta hipótesis será puesta a prueba en el apartado de experimentación de este trabajo.

3. Backtracking

La técnica de Backtracking consiste en ir generando solucion parciales, que se van extendiendo hasta llegar a una solución final, para enumerar todas las soluciones posibles. Es una idea similar a Fuerza bruta. De hecho el algoritmo presentando de Fuerza bruta sigue este idea, ya que Fuerza bruta se puede implementar con backtracking. La diferencia con Fuerza bruta radica en las denominadas *podas* que son reglas que permiten evitar explorar partes del árbol que genera backtracking, en las que se *sabe* que no va a existir ninguna solución de interés. Dicho de otra manera se *poda* el árbol, por lo que se tienen que analizar menos nodos. Generalmente estas podas dependen de cada problema en particular, pero las más comunes suelen dividirse en dos categorías: *factibilidad* y *optimalidad*. Para el caso del algoritmo propuesto en esta sección al problema de los jambo-tubos se utilizaron dos podas de factibilidad y una de optimalidad.

Podas por factibilidad La primera poda por factibilidad es la siguiente. Sea p el peso acumulado de los productos agregados en un nodo intermedio n_0. Si p ya es mayor a R entonces, no va a ver forma de extender la solución parcial a una solución válida debido a que todos los w_i son positivos, por lo que el peso que se agregaría extendiendo una solución sería siempre mayor a 0 (o 0 en caso de que no se agregue ningun producto más). Por lo tanto podemos retornar 0 ya que la solución ya no es válida ni puede serlo más adelante. Esta poda está expresada en la línea 3 del Algoritmo 3. La segunda poda por factibilidad es: Sea p el peso acumulado de los productos

agregados en un nodo intermedio n_0. Si existe un $producto_i$ que fue agregado y que no resiste el peso por encima de él (siendo el peso por encima de él, el peso total de los productos agregados hasta el nodo n_0 menos el peso del mismo producto i y de todos los productos que están por debajo de i), entonces esta solución parcial tampoco puede ser extendida a una solución factible, ya que por más que no se agregue ningún producto más, va a seguir sin resistir el peso, y si se agrega algún producto más como todos los pesos son positivos, va a tener más peso por encima, por lo que tampoco va a resistir. Se puede entonces retornar y devolver 0. Esta poda está expresada en la línea 4 del Algoritmo 3.

Poda por optimalidad Para el caso de esta poda, se utiliza otra variable auxiliar global denominada $maxValue$. En esta variable se irá almacenando la mejor solución encontrada hasta el momento por el algoritmo de backtracking, es decir la máxima cantidad de productos que se pueden apilar. Entonces para los nodos intermedios n_0, que representa a una solución parcial, se realiza la siguiente comprobación. En la variable k se tiene la cantidad de productos agregados para esa solución parcial. Si sumando a este valor, la cantidad de productos que no se revisaron aún (es decir, suponiendo que se pueden agregar todos los productos que faltan), no puedo superar al máximo encontrado, entonces no voy a llegar a una solución óptima por este subárbol. Podemos retornar 0 y así evitar el cómputo innecesario de operaciones.En el Algoritmo 3 se actualiza la variable $maxValue$ cada vez que se halla una mejor solución factible en la línea 7, y se evalúa la regla de la poda en la línea 5.

Algorithm 3 Algoritmo de Backtracking con podas para jambo-tubos.

```
1:  maxValue ← 0
2:  function BTPodas(i, k, p)
3:      if p > R then return 0
4:      if notTodosLosProductosResistenPeso(i, p) then return 0
5:      if k + (n − i) <= maxValue then return 0
6:      if i = n then
7:          if k > maxValue then maxValue ← k
8:          return k
9:      solucionParcial[i] ← false
10:     sinAgregar ← BTPodas(i + 1, k, p)
11:     solucionParcial[i] ← true
12:     agregado ← BTPodas(i + 1, k + 1, p + productos[i].peso)
13:     return máx{sinAgregar, agregado}.
```

El algoritmo presentado de backtracking es correcto debido a que es exactamente el mismo que el utilizado en fuerza bruta (que ya fue demostrado que es correcto porque genera todas las soluciones candidatas posibles) con el agregado de podas, que ya fueron explicadas por qué son correctas. Notar que cuando se llega a una hoja de este árbol generado ($i = n$) no es necesario preguntar si la solución encontrada es válida, ya que las podas se usan en todos los nodos (tanto los intermedios como los nodos hojas), por lo que si se llega a la condicion de que $i = n$, ya está garantizado que el peso acumulado p no supera la resistencia del tubo R y que todos los productos desde el primero hasta el último resisten el peso por encima, que es lo que validaba la función de $esSolucionValida$. La solucion al problema tambien se obtiene llamando a la funcion $BTPodas(0, 0, 0)$

La complejidad del algoritmo en el peor caso es $O(n * 2^n)$. Esto se debe a que el algoritmo genera el mismo árbol que fuerza bruta, pero ahora la diferencia es que en cada nodo, tanto intermedio como hoja, se llama a la funcion para verificar si todos los productos resisten el peso, que ya se demostró que tiene una complejidad temporal de $O(n)$ en el peor caso. La cantidad de nodos internos en un arbol completo binario de $n + 1$ niveles es $2^n - 1$ por lo que la cantidad de operaciones sería $(2^n - 1) * n$ para todos los nodos intermedios y $2^n * n$ para los nodos hoja. Por lo tanto la complejidad resultante es $O(n * 2^n)$. No se ha podido identificar igualmente un caso o un

tipo de instancias en particular en donde ninguna de las podas se utilice. Para los casos en donde ningún producto se puede agregar debido a que todos superan la resistencia del tubo, se resuelve de manera muy eficiente ya que la primera poda que verifica que el peso acumulado no supera la resistencia hace que no se revise ningun subárbol en donde se agregue un producto. Los casos en donde se agregan pocos porque muy pocos resisten el peso encima, también se resuelven de manera eficiente por la poda que verifica que si algún producto agregado ya no soporta el peso, entonces no se siga por ese subárbol. Y los casos en donde muchos productos se pueden agregar (o todos se pueden agregar) la poda de optimalidad es útil, ya que una vez que se encontró una solución donde se agregaron $maxValue$ productos, todas las instancias en donde no se van a agregar mas de $maxValue$ productos, ya no se revisan.

4. Programación Dinámica

La técnica de *Programación Dinámica* se utiliza cuando un problema cumple con la propiedad de superposición de subproblemas. Es decir, cuando en el cálculo recursivo de subproblemas de un algoritmo, hay muchos que se repiten. La idea consiste en evitar recalcular todo el subárbol correspondiente si ya fue hecho con anterioridad. Para esto, primero hay que definir una función recursiva que resuelva al problema y luego ver que se cumple la propiedad de superposición de problemas. La función recursiva propuesta para este problema de los jambo-tubos es la siguiente:

$$f(i,r) = \begin{cases} 0 & \text{si } i = 0, \\ f(i-1,r) & \text{si } i > 0 \wedge (w_i > r \vee r_i < R - r), \\ \text{máx}\{f(i-1,r), f(i-1,r-w_i)+1\} & \text{caso contrario.} \end{cases} \quad (1)$$

La semántica de la función $f(i,r)$ es la siguiente: "máxima cantidad de productos que se pueden apilar sin aplastarse del subconjunto de productos $\{S_1, \ldots, S_i\}$ dada la resistencia restante de peso del tubo r". Notar que el peso que soporta cada producto se puede calcular mediante la cuenta $R - r$ ya que la resistencia inicial de peso del tubo R, menos la resistencia que queda cuando se van agregando productos r, da como resultado el peso agregado. La solución al problema, dadas estas definiciones, es claramente $f(n,R)$. A continuación se dará una breve explicación de por qué la definición de esta función es correcta.

Correctitud

(i) Si $i = 0$ estamos buscando cuántos productos se pueden apilar de un conjunto vacío de productos, por lo que la respuesta es 0. Es el caso base.

(ii) Si $i > 0$ y $(w_i > r \vee r_i < R - r)$ entonces quiere decir que el peso del producto i supera a la resistencia restante del tubo, o que la resistencia del producto i es menor al peso que tiene por encima (que definimos que ese peso se puede calcular con $R - r$). En cualquiera de los dos casos, significa que el producto i no se puede agregar. Entonces, se devuelve el llamado de $f(i-1,r)$, es decir ese producto no se agrega.

(iii) En cualquier otro caso buscamos el máximo entre agregar o no el producto i, llamando a la misma función recursivamente, en un caso con los parametros $i-1$, y r, que significa que el producto no se agrega, por lo que resistencia restante del tubo sigue siendo la misma. Y en el otro caso se llama a la misma función con el parametro $i-1$ y el $r - w_i$, que significa que el producto se agregó, por lo que a la resistencia restante del tubo se le resta el peso del producto agregado. Notar que al término de la derecha se le suma 1 por haber agregado al i-ésimo producto. La recursión termina porque el parametro de i que empieza en n, en cada paso siempre se le resta 1, por lo que en algún momento llegará a 0, que es el caso base.

Memoización Para poder realizar la memoización de resultados es necesario verificar primero que se cumple la superposición de problemas. Si analizamos los parámetros que recibe la función (1) vemos que son $i \in [0, \ldots, n]$ y $r \in [0, \ldots, R]$. El caso cuando $i = 0$ es el caso base. Por lo tanto, la cantidad posible de argumentos distintos con los cuales se puede llamar a la función está determinada por la combinación de ellos, que en este caso son $\Theta(n * R)$. La función en peor caso puede invocarse a sí misma dos veces decrementando el parámetro i, por lo que la cantidad de llamados recursivos es $\Theta(2^n)$. Si $n * R << 2^n$ entonces se cumple la propiedad. Se concluye entonces que es beneficioso agregar una memoria que recuerde cuando un caso ya fue resuelto y su correspondiente resultado, para calcular una sola vez cada uno de ellos y asegurarnos no resolver más de $\Theta(n * R)$ casos. El Algoritmo 4 muestra esta idea aplicada a la función (1). En la línea 5 y 6 se lleva a cabo el paso de memoización que solamente se ejecuta si el estado no había sido previamente computado.

Algorithm 4 Algoritmo de Programación Dinámica jambo-tubos.

```
1: M_ir ←⊥ for i ∈ [0, n], r ∈ [0, R].
2: function DP(i, r)
3:     if i = 0 then return 0
4:     if M_ir =⊥ then
5:         if productos[i − 1].peso > r or productos[i − 1].resistencia < R − r then M_ir = PD(i − 1, r)
6:         elseM_ir =max{PD(i − 1, r), PD(i − 1, r − productos[i − 1].peso) + 1}
7:     return M_ir
```

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 $O(n * R)$ estados distintos, y como todas las líneas del Algoritmo 4 realizan operaciones constantes entonces cada estado se resuelve en $O(1)$. Como resultado, el algoritmo tiene complejidad $O(n * R)$ 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 $\Theta(n * R)$, por lo tanto, el mejor y peor caso de nuestro algoritmo va a tener costo $\Theta(n * R)$.

5. Experimentación

En esta sección se presentarán experimentos computacionales realizados para evaluar los distintos métodos presentados en las secciones anteriores y probar ciertas hipótesis. Los algoritmos utilizados en esta sección son **FB** (Algoritmo 1 de Fuerza Bruta de la Sección 2), **BTPodas** (Algoritmo 3 de Backtracking de la Sección 3) y **PD** (Algoritmo 4 de Programación Dinámica de la Sección 4).

5.1. Instancias

Para evaluar los algoritmos en distintos escenarios es preciso definir familias de instancias conformadas con distintas características. Los algoritmos se han ejecutado con casos en donde se podía agregar todos los productos y casos en donde no se podía agregar ningún producto. También se han realizado casos en donde el R es muy grande, como también R aleatorio y resistencia de productos aleatorios. Las instancias que se utilizaron están detalladas a continuación:

- **ningun-producto-agregable**: En este tipo de instancias todos los productos S_i de $S = \{1, \ldots, n\}$ con $n <= 30$ tienen un peso $w_i > R$ siendo $w_i = 1001$ y $R = 1000$. Además ningún producto resiste el peso de otro producto encima. Los r_i se definieron en orden creciente empezando en 1 para S_1 y terminando en n para S_n, con incrementos de a una unidad ($r_i = i$). Es decir en este tipo de instancias no se puede agregar ningún producto, por la que respuesta al problema será siempre 0.

- **todos-los-productos-agregables**: En este tipo de instancias la resistencia del tubo R es 1000. Todos los productos S_i de $S = \{1, \ldots, n\}$ con $n <= 30$ tienen un peso $w_i = i$, es decir los pesos de los productos comienzan en 1 para S_1 hasta n para S_n, con incrementos de a una unidad. Todas las resistencias de los productos son iguales ($r_i = 1000$). Se tiene que $\sum_{i=1}^{n} w_i < R$ y que $\forall S_i(\sum_{i=i+1}^{n} w_i) < r_i$ con $1 <= i < n$. Por lo tanto, todos los productos se pueden agregar en este tipo de instancias, ya que todos resisten el peso por encima y la suma de todos sus pesos es menor a la resistencia de tubo. La respuesta al problema en estos casos, es siempre n.

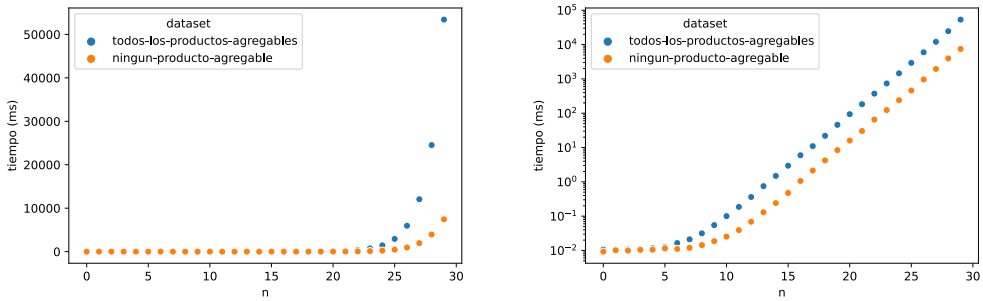
- **R-grande**: El valor de R para todos estos tipos de instancias es 400000000. Se generan también instancias que contienen hasta $n = 30$ productos. Los valores de los pesos y la resistencias para cada S_i son aleatorios. $\forall S_i w_i \in [1, 500]$ y $r_i \in [1, 500]$. Se utilizarán principalmente para estudiar el algoritmo de programación dinámica.

- **random**: Se generan instancias con hasta $n = 30$ productos con valores aleatorios. El $R \in [500 - 20000]$ y $\forall S_i w_i \in [1, 500]$ y $r_i \in [1, 500]$. La idea de utilizar un R con valores aleatorios que estén comprendidos en un rango de valores un poco mayor que w_i y r_i es para permitir que algunos productos se puedan agregar.

- **random-muchos-productos**: Esta familia de instancias es exactamente igual a la anterior, con la salvedad de que se generan instancias con hasta $n = 140$ productos. Utilizada para ciertas hipotesis con respecto al algoritmo de backtracking con podas.

5.2. Experimento 1: Complejidad y analisis de Fuerza Bruta

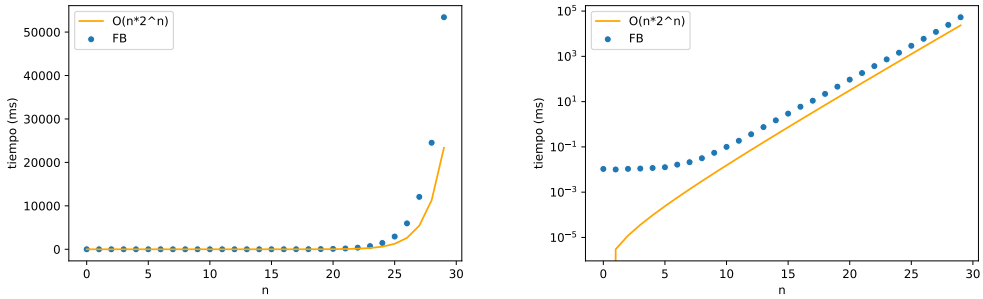
El objetivo de este experimento es comprobar empíricamente la complejidad teórica del algoritmo de fuerza bruta (FB), como asi analizar ciertos casos para observar su comportamiento según el tipo de instancia sobre el cual se ejecuta. La hipótesis es que para las instancias mencionadas *ningun-producto-agregable* posea un tiempo de ejecucion menor que para las instancias del tipo *todos-los-productos-agregables*. La razón de esta hipótesis es la construcción del algoritmo de FB, específicamente en su sección de verificación de si una solución es válida o no, que por como está implementada, no itera siempre hasta el final de la lista de productos, sino que corta el ciclo ni bien encuentra un producto que no resiste el peso (linea 6 de Algoritmo 2). La Figura 2 presenta los resultados del experimento en donde se comparan los tiempos de ejecución del algoritmo de FB para las instancias de *ningun-producto-agregable* y *todos-los-productos-agregables* en función del tamaño de la entrada n. Se puede observar que el algoritmo efectivamente se comporta de mejor manera en términos de tiempo de ejecución para las instancias de *ningun-producto-agregable*, a partir de $n = 5$ aproximadamente, por lo que se pudo probar la hipótesis, si bien la diferencia entre ambos tipos de instancias no es muy significativa. Es decir, se concluye que es una buena idea no iterar hasta el final de la lista de productos, si es que ya se encuentra uno inválido porque puede reducir los tiempos de ejecución. La otra figura 3 son datos ejecutados sobre la instancia de *todos-los-productos-agregables*. Se usaron también las escalas logarítmicas para apreciar mejor la diferencia de tiempos de ejecución para instancias de menor tamaño. Se puede concluir a partir de los gráficos que efectivamente los tiempos de ejecución se corresponden con la cota de complejidad teórica dada de $O(n * 2^n)$. En la escala logarítmica se aprecia claramente una recta, a partir de instancias de tamaño 5, que por ser la escala logarítmica, significa que crece de manera exponencial. El índice de correlación de Pearson entre la cota de complejidad teórica y el tiempo de ejecución para la instancia de *todos-los-productos-agregables* es positivo y es $r \approx 0{,}99979$, como se ve en la figura 4.



(a) Tiempo de ejecución de FB sobre cantidad de productos

(b) Tiempo de ejecución de FB sobre cantidad de productos (escala logaritmica)

Figura 2: Comparacion de FB con distintos tipos de instancias



(a) Tiempo de ejecución de FB sobre cantidad de productos

(b) Tiempo de ejecución de FB sobre cantidad de productos (escala logaritmica)

Figura 3: Comparacion de FB con cota teorica

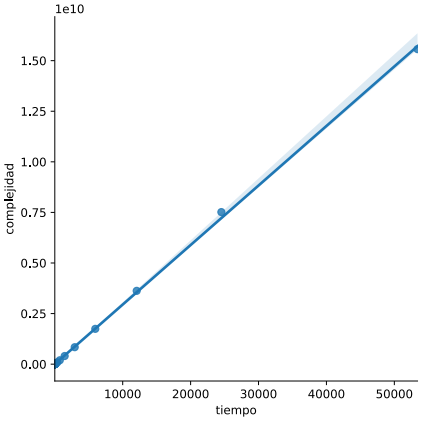
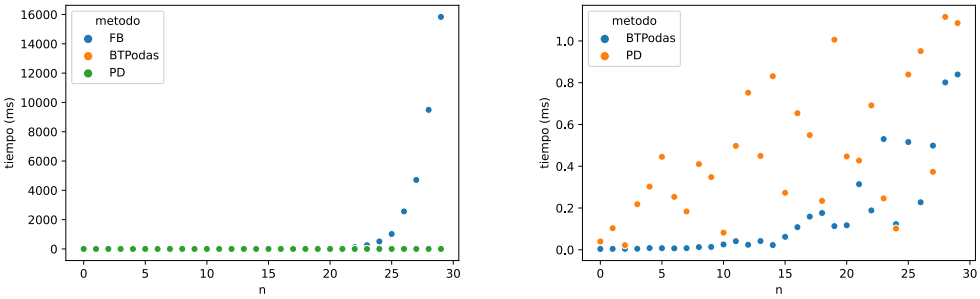


Figura 4: Indice de correlacion de Pearson de FB con cota teorica

5.3. Experimento 2: Complejidad de los algoritmos (pocos productos)

En esta experimentación vamos a comparar las complejidades de los tres algoritmos, en donde tomamos desde 1 a 30 productos. La resistencia del tubo varía de 500 a 20000, tomado en forma

aleatoria, como también el peso y la resistencia de los productos, variando de 1 a 500. Se uso funciones de la biblioteca random de python para generar los datos. Nuestra hipótesis es que la fuerza bruta será el menos eficiente y la programación dinámica el mas eficiente, por los cálculos de complejidad teórica realizadas en los incisos anteriores.



(a) Tiempo de ejecución de los metodos Fuerza bruta, Backtracking y Programación dinámica sobre cantidad de productos

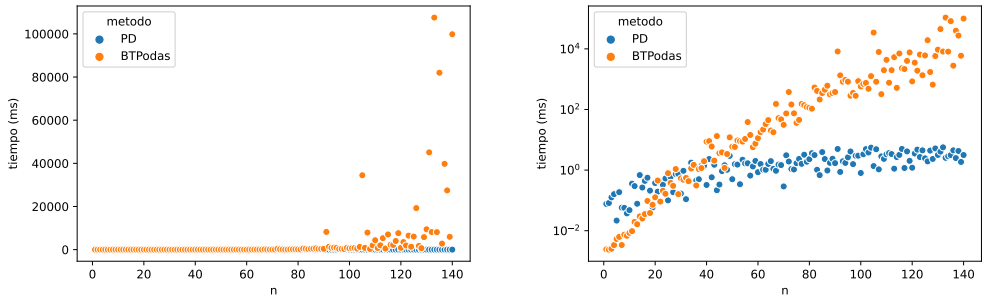
(b) Tiempo de ejecución de los metodos Backtracking y Programación dinámica sobre cantidad de productos

Figura 5: Análisis de complejidad de los tres métodos

En la figura 5a se puede ver que el algoritmo de fuerza bruta tarda mucho mas que los otros dos algoritmos, de manera que los otros dos algoritmos son indistinguibles a simple vista. Por lo cual, se hizo la figura 5b, donde se comparó solamente el backtracking con la programación dinámica. Se puede ver que con un R acotado y una cantidad de productos relativamente acotado (de hasta 30 productos), el backtracking parece ser más eficiente en promedio que la programación dinámica. Este resultado pareciera refutar nuestra hipótesis! Pero, tenemos que recordar que las complejidades teóricas tienen implícitamente una constante multiplicativa y se vale a partir de cierto n en adelante. Por lo cual, sólo podemos concluir que en nuestro caso de test particular, el bactracking tiende a ser el más eficiente de los tres algoritmos.

5.4. Experimento 3: Complejidad de los algoritmos (muchos productos)

Naturalmente, surgen varias preguntas luego del experimento anterior. En particular, ¿siempre es mas eficiente el backtracking? ¿La complejidad teórica no es válica? Ahora, nuestra hipótesis es que para relativamente pocos productos y con un R relativamente chico, el backtracking tiene ventaja. Pero cuando mantenemos el R en el mismo rango aumentando la cantidad de productos, a partir de cierto momento la programación dinámica será mas eficiente.



(a) Escala normal

(b) Escala logaritmica

Figura 6: Comparación del tiempo de ejecución del backtracking y de la programación dinámica, de hasta 140 productos, con su peso y resistencia sampleado aleatoriamente.

En definitiva, como se puede ver en la figura 6a, a partir de cierta cantidad de productos, el backtracking comienza a tardar cada vez más tiempo para ejecutar. Esto indica que el beneficio de guardar en memoria los subproblemas ya resueltos comienza a mejorar notablemente la eficiencia, que corrobora con nuestra hipótesis. Para esclarecer mas, se hizo una escala logarítmica, como se puede ver en la figura 6b, para determinar con más exactitud desde qué cantidad de productos en adelante comienza a aventajar la programación dinámica. Observamos que con $n = 50$ en adelante, el backtracking ya no es el algoritmo mas eficiente, y, no sólo eso, sino que la diferencia aumenta exponencialmente. Entonces, para un R tal que $nR << 2^n$, y un n suficientemente grande, la programación dinámica tiene una complejidad mucho menor al bactracking.

5.5. Experimento 4: Fuerza bruta vs Programación Dinámica

El último experimiento que se presentará en este trabajo es una comparación particular de PD y FB. Para este experimiento se usaron las intancias de *R-grande* para los dos algoritmos. La hipótesis en este experimiento es que para instancias en donde el R es muy grande (en este caso es igual a 400000000), PD sea incluso menos eficiente en tiempos de ejecución que FB para instancias pequeñas. De la figura 7 se puede observar que PD presenta tiempos de ejecución superiores para instancias chicas, comprobando la hipótesis. Esto se debe en parte a que hay que inicializar una matriz de un tamaño bastante grande $n * R$ que en FB no hay que hacerlo. El crecimiento se puede ver que es lineal para PD. Eso se debe a que crece también cuando el número de productos aumenta (mantieniendo el R constante). Esta observación igualmente es para instancias en donde el n se mantiene en un rango no muy alto, ya que cuando n crece mucho, la función exponencial que representa el tiempo de ejecución de FB va a crecer lo suficiente como para *ganarle* al R grande. Por lo tanto, en caso de tener un número de productos pequeño pero con un R muy alto, PD no es una buena opción, incluso usar FB, que es una de las técnicas menos eficientes, es mejor.

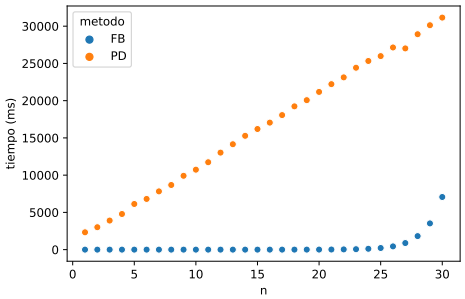


Figura 7: FB vs PD en función de cantidad de productos con R = 400000000

6. Conclusiones

Se utilizaron tres técnicas de programación distintas para resolver el problema presentado de los jambo-tubos. Del análisis y la experimentación realizada se puede concluir que Fuerza bruta es un algoritmo bastante ineficiente y solamente se puede aplicar para instancias muy chicas (con pocos productos en este caso) ya que para instancias grandes su tiempo de ejecución no es aceptable. La técnica de Backtracking con podas es bastante útil y para instancias con pocos productos es de hecho más rápido que programación dinámica. No obstante, cuando el tamaño de la entrada crece un poco, ya empieza a tener los mismos problemas que Fuerza bruta por más que se utilicen podas. La técnica de programación dinámica es una de las mejores para este problema, ya que presenta muy buenos tiempos de ejecución para incluso instancias con un número de productos importantes, pero es dependiente no sólo de la cantidad de productos, sino también de R, y cuando el R crece de manera muy considerable, puede ser incluso menos eficiente que Fuerza Bruta. Otras de las desventajas de la programación dinámica es que ocupa mucha más memoria: $O(nR)$ para ser exacto. Lo que significa que en caso de que n es lo suficientemente grande, podría llegar a tener problemas de espacio de memoria, aunque el análisis de este caso no está dentro del alcance de este TP.

Como trabajo futuro, se puede pensar en una implementación de Programación dinámica con un enfoque $bottom - up$ iterativo para analizar si se puede mejorar más los tiempos de ejecución, disminuyendo la sobrecarga de la recursión, y disminuir la complejidad espacial del algoritmo, que es mayor al de los otros dos.