DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP1 Optimizando Jambo-tubos

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Venegas Ramírez, David Alejandro	783/18	davidalevng@gmail.com
Bustamante, Luis Ricardo	43/18	luisbustamante097@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

Los problemas de *optimización combinatoria* son un área de estudio central en las ciencias de computación con importantes aplicaciones en la industria y otras ciencias. En el presente trabajo se resuelve optimizar una variante del conocido problema de la *mochila 1-0* (Knapsack Problem 1-0 en inglés) aplicado al empacado por robots de productos en recipientes cilíndricos denominados jambo-tubos.

Formalmente, dado un Jambo-Tubo de resistencia máxima $\mathbf{R}$, y una secuencia ordenada de $\mathbf{n}$ productos $\mathbf{S} = \{\mathbf{s_0}, ..., \mathbf{s_{n-1}}\}$, cada uno con un peso asociado $\mathbf{w_i}$ y una resistencia asociada $\mathbf{r_i}$, se quiere determinar la máxima cantidad de productos que pueden apilarse en un tubo sin que ninguno sea aplastado, es decir, la combinación óptima de elementos para guardar en el tubo. Por ende, se debe implementar un algoritmo que tome en cuenta el peso de cada ítem, la resistencia de los mismos y del jambo-tubo para encontrar la mejor combinación de elementos, siendo la resistencia el peso máximo que puede soportar un elemento sin romperse.

A continuación, se exhibe una ejemplificación del problema en cuestión: Dado un tubo de resistencia 50, y 5 ítems $S = \{s_0, s_1, s_2, s_3, s_4\}$ los cuales tienen su peso y resistencia correspondiente en las siguientes listas respectivamente: $w = [10, 20, 30, 10, 15]$ y $r = [45, 08, 15, 02, 30]$. El resultado buscado es 3 que deriva del empaquetado de los ítems s_0, s_2 y s_3, ya que $10 + 20 + 30 \leq 50$ tal que ningún ítem aplasta al anterior ni se rompe el jambo-tubo.

Se plantean 3 técnicas algorítmicas distintas para resolver el problema: *Fuerza Bruta* que prueba todas las posibles combinaciones y selecciona la solución óptima si existe, *Backtracking* semejante a FB pero que reduce la cantidad de combinaciones a computar mediante recorte de los llamados recursivos y *Programación Dinámica* que permite guardar progresivamente los resultados obtenidos a través de una memoización de los mismos para re-utilizarlos sin incurrir en cómputos extra. A lo largo del presente proyecto se considerarán las instancias beneficiosas o adversarias para estos algoritmos y como se comparan ante la necesidad de resolver el problema de los jambo-tubos.

2. Fuerza Bruta

Para el presente problema en particular dado un conjunto S de entrada, se recorre todo el conjunto de partes de S, llamado $\mathcal{P}(S)$. Retomando el ejemplo anterior (con un elemento menos), con $R = 50$, y sea $S = \{s_0, s_1, s_2, s_3\}$, $w = [10, 20, 30, 10]$ y $r = [45, 08, 15, 02]$ listas que representan el peso y la resistencia correspondiente a cada uno, se puede apreciar que el conjunto resultado va a ser el subconjunto $[s_0,s_2,s_3]$ como lo muestra la Figura 1. Además, se observa que es el único subconjunto que cumple los requisitos descriptos en la introducción.

A continuación se va a presentar el pseudo-código de Fuerza Bruta implementado. Nótese que para obtener el resultado final se debe llamar a $FB(w, r, 0, R)$.

Algorithm 1 Algoritmo de Fuerza Bruta.

```
1: function FB(w, r, i, min_r)
2:     if i = n then
3:         if min_r < 0 then return −∞
4:         return 0
5:     return mín{FB(w, r, i + 1, min_r), FB(w, r, i + 1, mín{r[i], min_r − w[i]}}.
```

En la Figura 1 se representa el árbol de recurrencia descripto por la ejecución del Algoritmo 1, donde cada nodo intermedio representa una solución parcial y los hijos de cada nodo representa el haber incluido o no a un elemento del conjunto S. Por lo tanto las hojas representan a todas las soluciones, es decir el conjunto de partes de S.

La primera fila debajo de las hojas representa el cardinal del subconjunto resultado. Se puede ver que los mayores números son el 3 y el 4, por ya se conoce que los de cardinal menor no pueden ser solución óptima. Y en la segunda fila se ve el peso acumulado por cada solución, se sabe que la solución final no puede ser de mayor peso que R. Por último se ve en verde la hoja que cumple

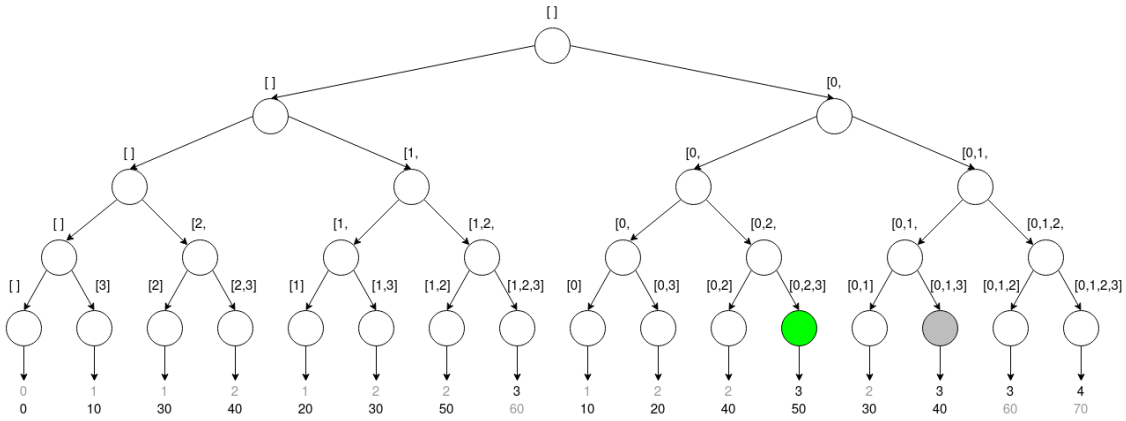


Figura 1: Ejecución del Algoritmo 1 para un conjunto S de 4 elementos, $w = [10, 20, 30, 10]$, $r = [45, 08, 15, 02]$ y $R = 50$
La primera fila de texto describe el cardinal de las soluciones, y la segunda el peso acumulado. En verde está la solución final, y en gris una solución que no cumple los requisitos.

los requisitos para ser el resultado final del Algoritmo 1. Y en gris se ve a un nodo que tiene el peso buscado, pero se puede comprobar que algún elemento estaría aplastado.

Correctitud: La correctitud del algoritmo toma el hecho de que se recorre todo el conjunto de partes de S, por lo tanto siempre se va a encontrar la solución final si existe. El criterio de elección para la solución óptima es tomar al *máximo cardinal* del subconjunto que cumpla que: la mínima resistencia posible de la instancia siguiente va a ser el mínimo entre la resistencia del elemento mas frágil hasta ese nodo (fíjese que esta decrementa medida que se agregan objetos) y la capacidad máxima del tubo, y por último que la suma de los pesos del subconjunto no supere a R.

Complejidad: Dado que el algoritmo recorre todos los casos posibles de subconjuntos de S, es decir que analiza cada subconjunto de $\mathcal{P}(S)$ el cual tiene 2^n elementos, y en vista de que las operaciones hechas en cada paso son operaciones elementales, incluido *min* que se toma por una función que tiene complejidad $O(1)$ (desde el punto de vista uniforme); se puede concluir que la complejidad final del algoritmo es $O(2^n)$, y en particular va a ser $\Theta(2^n)$ ya que aun en el mejor caso va a realizar todas las comparaciones.

3. Backtracking

El algoritmo de backtracking aprovecha la estructura recursiva de FB pero agrega consideraciones especiales que permiten disminuir la cantidad de soluciones a explorar y en el caso promedio tiene un rendimiento superior; aunque mantiene la misma cota de complejidad teórica que FB. Este algoritmo genera un *árbol de backtracking* con todas las posibles combinaciones a ítems para guardar en el jambo-tubo; cada nodo o rama del árbol contiene las soluciones locales (de la cual se puede seleccionar la mejor hasta el momento) y sobre estas ramificaciones se aplican criterios para decidir si se debe explorar una rama para encontrar la solución óptima.

Esta serie de reglas son conocidas como *podas*, y aunque cada poda depende del problema en particular a resolver, se suelen generalizar en *podas por factibilidad* y *podas por optimalidad*.

Poda por factibilidad: Evita evaluar casos donde no es posible encontrar una solución, en esta caso es la siguiente: sea min_r la resistencia mínima de un jambo-tubo en S', es decir una solución parcial donde dado que el peso de cada ítems es positivo si $min\_r < 0$ entonces no existe ninguna extensión de S' que sea solución (ni esta lo es) ya que al agregar otros ítems min_r sólo decrecerá .

Algorithm 2 Algoritmo de Backtracking para Jambo-tubos.

```
1:  MS ← −∞
2:  function BT(w, r, i, cant_hasta, min_r)
3:      if i = n then
4:          if min_r < 0 then return −∞
5:          return cant_hasta
6:      if min_r < 0 then return −∞                          ▷ Poda por factibilidad
7:      bool_aux ← if min_r ≥ 0 then cant_hasta + 1 + min_r ≤ MS else false
8:      if bool_aux | | cant_hasta + n − 1 ≤ MS then          ▷ Poda por optimalidad
9:          if min_r < 0 then return −∞ else return cant_hasta
10:     if min_r ≥ 0 then MS ← máx(cant_hasta − 1, MS)
11:     return máx{BT(w, r, i + 1, cant_hasta, min_r),
12:                BT(w, r, i + 1, cant_hasta + 1, min(r[i], min_r − w[i]))}.
```

Así se evita explorar un subárbol donde no existe solución y se disminuye la cantidad de cómputos a realizar. Esta poda está expresada en la línea 6 del Algoritmo 2.

Poda por optimalidad: Dado que para un problema pueden existir muchas soluciones factibles, se plantea para una solución parcial S' con k elementos, un nodo intermedio n_0 de dicha solución donde se puede evaluar si los caminos a explorar devendrán en una solución óptima. Para esto se guarda la variable global MS que es actualizada cuando se encuentra un solucion valida y mejor que la ya conocida hasta el nodo n_0. En este caso, si $cant\_hasta + min\_r \leq MS$, se puede asegurar que cualquier solución factible que se encuentre al explorar dicho subárbol no va a ser mejor que la que ya encontrada y por ende no se exploran los nodos subsiguientes. Esto se debe a que cualquier decisión que se tome a continuación en el subárbol implica agregar o mantener los ítems seleccionados, y como el peso de cada elemento es positivo, entonces la mínima resistencia limita en el mejor caso el total de elementos a agregar antes de que sea menor a 0. De forma análoga, sucede lo mismo para el caso cuando $cant\_hasta + |S'| - i < MS$ ya que aún agregando todos los elementos restantes del subárbol no es posible encontrar una mejor solución . En definitiva, se puede podar las rama según estos criterios de forma eficiente y segura para evitar el cómputo innecesario de subproblemas. En el Algoritmo 2 se utiliza MS en la línea 7 para actualizar el valor de la mejor solución factible, y en la línea 8 se realiza la poda correspondiente.

Complejidad: Se mantiene complejidad del algoritmo de FB en el peor caso DE $O(2^n)$ ya que en un escenario para una instancia no favorable donde no se logre realizar podas se terminará explorando todas las posibles combinaciones de ítems para guardar en los jambo-tubos, es decir, se enumera todo $\mathcal{P}(S)$. Además, se puede observar que el código introducido en las líneas 7, 6 y 8 solamente agrega un número constante de operaciones en $O(1)$.

Estudiando cada poda por separado se evidencia que en el caso de las podas por factibilidad existe una familia de instancias las cuales disminuyen la eficiencia de este algoritmo y aumentan los nodos que se deben recorrer en el árbol de recursiones. En particular, cuando la suma de los pesos de todos los ítems de la cinta transportadora es menor a la resistencia del jambo-tubo en cuestión y en especial, cuando al colocar todos los elementos, ninguno se rompe. Mejor expresado de la siguiente forma:

$$\sum_i w_i \leq R \wedge (\forall i : \mathbb{N})(0 \leq i \leq |w| \implies (\forall j : N)(i < j < |w| \implies r_i \geq \sum_j w_j)) \tag{1}$$

En cambio, no es posible realizar podas por optimalidad cuando el problema no tiene solución y nunca se agrega algún elemento, es decir, cuando tenemos que

$$(\forall i : \mathbb{N})(0 \leq i < |w| \implies w_i < R). \tag{2}$$

Sin embargo, durante la investigación no se encontraron instancias que anulen en su totalidad ambas podas en simultáneo y obliguen al algoritmo a recorrer todos los nodos. Esto se debe a que el primer llamado recursivo siempre agrega un elemento y si existe una solución para el problema esta se va a guardar como una variable global MS y por ende se van a realizar podas por optimalidad, pero si no existiese solución, es decir sino se pudiesen hacer podas por optimalidad, en ese caso, toman inmediato efecto las podas por factibilidad y no se exploran todas las partes de S. Esto es un indicio de que la complejidad final del algoritmo puede ser menor a la cota planteada y queda como base para futuras investigaciones. Sin embargo, mas adelante en la experimentación queda claro que existe una familia de instancias exponenciales para dicho algoritmo.

El mejor caso del algoritmo de backtracking ocurre cuando gracias a las podas se recorre una sola vez cada nodo, y por ende termina en tiempo lineal $O(n)$. Esto ocurre en 3 instancias genéricas: cuando no hay solución gracias a las podas por factibilidad, cuando la solución es un único ítem distinguible para agregar (pues todos los demás tienen un peso mayor a la resistencia del jambo-tubo), o cuando todos los ítems del problema entran en el tubo. En este último caso toman efecto las podas por optimalidad que una vez que encuentran la mejor solución garantizan que en ningún otro nodo se va a ramificar, y como el primer llamado recursivo siempre agrega un elemento, la mejor solución es la primera en encontrarse y en consecuencia las ramificaciones de todos los demás nodos no se exploran.

4. Programación Dinámica

Se acaban de presentar dos técnicas que van recorriendo las posibles combinaciones de elementos, pero en ambas se superponen subproblemas que son computados múltiples veces al calcular resultados intermedios para llegar al caso base. Entonces, uno se podría preguntar: *¿Existe algún modo de poder guardar cada uno de los resultados que ya se hayan hecho?*. Bajo esta pregunta entran los algoritmos de *Programación dinámica*, donde si se sabe que existe *superposición de subproblemas*, se puede resolver el problema con dicha técnica algorítmica. Que haya superposición de problemas significa que en una solución óptima, cada subsolución es a su vez, óptima de su subproblema correspondiente. Pero primero se definirá la función recursiva que resuelve el problema:

$$f(i,m) = \begin{cases} -\infty & \text{si } m < 0, \\ 0 & \text{si } m = 0, \\ 0 & \text{si } i = n, \\ \text{máx}\{f(i+1,m), 1 + f(i+1, \text{mín}\{r_i, m - w_i\}) & \text{caso contrario.} \end{cases} \tag{3}$$

La ecuación coloquialmente está diciendo que $f(i,m)$ es igual a "la máxima cantidad de elementos $\{S_i, \ldots, S_{n-1}\}$ que se puede poner en el tubo cumpliendo el orden, si m es el máximo peso que puede cargar el jambo-tubo sin que se rompa el mismo o alguno de sus elementos".

Correctitud

(i) Si $m < 0$ significa que el tubo superó su capacidad máxima y colapsó, por lo que no es una solución factible, entonces la respuesta es $f(i,m) = -\infty$.

(ii) Si $m = 0$ dice que el tubo ya no soportará un nuevo elemento, entonces ya sea que $i = n$ o $i \neq n$ la cantidad de elementos a colocar será 0, es decir que $f(i,m) = 0$

(iii) Similar al anterior ítem, si $i = n$ significa que no hay mas elementos para colocar, por lo que la respuesta es $f(i,m) = 0$

(iv) Ya que ahora $i < n$ y $m > 0$ se necesita encontrar la forma de retornar el máximo cardinal tal que se cumpla la premisa, se puede separar el análisis en dos ramas, si se coloca el ítem i en el tubo o si no. Si no se coloca se debe seguir analizando que pasa al agregar el siguiente

ítem, es decir que se calcula $f(i+1, m)$, nótese que la solución de ese subproblema es parte de la solución, por esa razón se puede calcular de forma recursiva. Ahora la parte interesante pasa si se incluye el ítem i al tubo, se debe incrementar el resultado en 1 ya que se sabe que el tubo tiene un elemento más apilado (no interesa si ya habían otros antes), y además se suma a eso la máxima cantidad de elementos que se puede agregar de los que quedan todavía. Considerando que el ítem i ya agregó un peso extra al tubo, la nueva resistencia máxima será el mínimo entre la resistencia del ítem i y la vieja resistencia máxima menos el peso de i, es decir el mín$\{r_i, m - w_i\}$. Así que se puede concluir que este caso es igual a $1 + f(i+1, \text{mín}\{r_i, m - w_i\}$. Dicho todo esto, la respuesta será el máximo entre los resultados de ambas ramas, o sea $f(i, m) = \text{máx}\{f(i+1, m), 1 + f(i+1, \text{mín}\{r_i, m - w_i\})$.

Memoización: Retomando lo mencionado al inicio de esta sección, se expuso que si un problema tenía *superposición de subproblemas* se podía hacer uso de esta técnica algorítmica que consiste en guardar el resultado de cada uno de los resultados que se va obteniendo en alguna estructura M que permita acceder rápidamente al dato, pero antes es necesario saber las dimensiones de esta estructura. Algo que ya se sabe es que una de las dimensiones va a ser $n - 1$ ya que es la cantidad de objetos que se quiere guardar, por otro lado, la función puede tomar hasta un $m = R$, siendo R la resistencia máxima del tubo, aunque cabe preguntarse qué pasa cuando $m > R$; ahora bien, viendo a los casos bases de la función, se observa que en este caso la función ya está definida. Con todo esto, ya se tiene la dimensión de la estructura M la cual va a ser $(n-1) * R$ para poder albergar hasta cada una de las combinaciones entre los parámetros i y m y se inicializan todos los valores en -1. En el algoritmo 3 se muestra el uso de la técnica, particularmente en la línea 6 que hace uso de la estructura de almacenamiento.

Algorithm 3 Algoritmo de Programación Dinámica para Jambo-Tubos.

1: $M_{im} \leftarrow \perp$ for $i \in [0, n-1], m \in [0, R]$.
2: **function** $DP(i,\ r)$
3: **if** $m < 0$ **then return** $-\infty$
4: **if** $m = 0$ **then return** 0
5: **if** $i = n$ **then return** 0
6: **if** $M_{im} = \perp$ **then** $M_{im} \leftarrow \text{máx}\{DP(i+1, m), 1 + DP(i+1, \text{mín}\{r_i, m - w_i\})\}$
7: **return** M_{im}

Complejidad: Desde la línea 1 tenemos que la complejidad del algoritmo será de al menos $\Omega(n * R)$ ya que tal línea inicializa cada casillero de M con un valor indefinido $\perp$ Acá se hace un paréntesis, ya que en la implementación de este informe se trabaja con M una matriz de $(n-1)*R$, pero se puede utilizar cualquier estructura de tipo diccionario que permita un acceso y escritura rápido. Entonces como la matriz tiene dimensión $(n-1) * R$ y la función solo está definida para $0 \leq i < n$ y $0 \leq m \leq R$ donde cada iteración va incrementando a i hasta que llegue a un caso base, y donde las operaciones intermedias son $O(1)$ (incluidas las funciones min y max), se puede concluir que la complejidad del algoritmo en el peor caso es $O(n * R)$ y en particular como la cota inferior para cualquier caso es sabida, entonces se concluye que la complejidad del algoritmo 3 es $\Theta(n * R)$ para cualquier caso.

5. Experimentación

A continuación se realiza un análisis comparativo del rendimiento experimental de cada técnica algorítmica. En cada caso de estudio se toma en cuenta el tiempo de cómputo y su correlación con la complejidad teórica señalada, para luego evaluar instancias particulares de interés con cada algoritmo. Es decir, se evalúa que conjuntos de ítems son beneficiosos o adversos a la eficacia de cada técnica.

En el presente análisis se utilizó una workstation con CPU Intel Core i7 @ 2.2 GHz y 8 GB de memoria RAM, y el lenguaje de programación *C++* en conjunto con jupyter-notebooks para generar las instancias y visualizar la data luego de correr los experimentos.

5.1. Métodos

Las técnicas utilizadas 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 8, es decir, solamente aplicando podas por factibilidad.
- **BT-O**: Similar al método BT-F pero solamente aplicando a podas por optimalidad, o sea, descartando la línea 6 del Algoritmo 2.
- **DP**: Algoritmo 3 de Programación Dinámica de la Sección 4.

5.2. Instancias

Para efectuar un análisis comparativo (o *benchmarking* en inglés) apropiado es importante definir familias de instancias adecuadas, que permitan modelar los escenarios de interés para el algoritmo, en especial en sus extremos (mejor o peor caso).

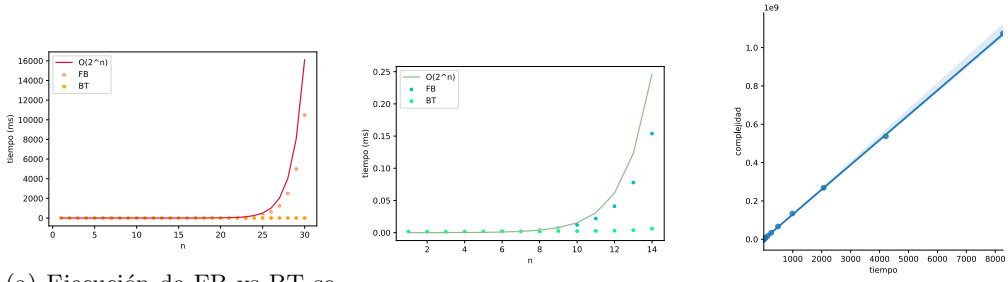
En pro de determinar los distintos escenarios de interés es preciso plantear una hipótesis que será luego corroborada mediante la experimentación. De esta forma, para definir los *datasets*, se plantea la variable *densidad* de una instancia como el cociente $\frac{\sum r_i}{\sum W_i}$, es decir, es una medida o *ratio* de cuantos grupos de elementos son solución, una mayor densidad implicará que menos elementos conforman parte de la solución, pues sus pesos son más grandes con respecto a sus resistencias, el caso contrario es análogo.

Finalmente, los *datasets* definidos se enumeran a continuación.

- **densidad-alta**: En esta familia cada instancia tiene los números $1, \ldots, n$ en w en algún orden aleatorio y valores desde 1 hasta $\frac{n}{2}$ en r.
- **densidad-baja**: Para este conjunto de instancias se toman los números $1, \ldots, n/2$ en w en algún orden aleatorio y desde 1 hasta n en r.
- **bt-mejor-caso**: Cada instancia de n elementos, está formada por $w = \{R+1, \ldots, R+1, R-1\}$ las instancias para el mejor caso de Backtracking definidas en la Sección 3.
- **bt-peor-caso-optimalidad**: Cada instancia de n elementos, está formada por $w = \{R+1, \ldots, R+1\}$. Son las instancias para el peor caso de Backtracking definidas en la Sección 3.
- **bt-peor-caso-factibilidad**: Cada instancia de n elementos, está formada por $w = \{a, ..., a\}$ tal que $\frac{R}{a} > n$ y $r = \{R, \ldots, R\}$. Son las instancias para el peor caso de Backtracking definidas en la Sección 3.
- **dinámica**: En esta familia de instancias se presentan varias combinaciones de n y r en los intervalos $[1000, 8000]$. Los números en r son una permutación de el conjunto $\{1, \ldots, n\}$. Para el caso de la comparación con BT se utilizan los valores de n entre 2 y 30 y $R = 500 o R = 1000$ con resistencias variables entre 100 y 200.
- **bt-mal-caso**: Cada instancia de n elementos (n par) en su primera mitad tiene $\frac{n}{2}$ elementos que conforman la solución de su problema, es decir cuya suma de pesos es menor a R, y en la otra mitad no va a tener ningún elemento que deba pertenecer al tubo pues su peso es mayor a R. Este caso particular minimiza las podas y genera una complejidad exponencial.

5.3. Experimento 1: Complejidad de Fuerza Bruta vs Backtracking

En el siguiente experimento se analiza el tiempo de los algoritmos de *FB* (Fuerza Bruta) y *BT* (Backtracking) vistos en las sección 2 y 3 respectivamente, frente al set de instancias *alta-densidad*



(a) Ejecución de FB vs BT sobre densidad-alta y superposición de FB con $f(n) = 2^n$

(b) Ejecución de FB vs BT sobre densidad-alta con zoom

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

Figura 2: Comparación entre FB y BT, junto a análisis de BF.

en la Figura 2. Y a la vez se hace hincapié a la correlación entre los tiempos de FB y su complejidad esperada $O(2^n)$.

En la Figura 2a se aprecia la gran diferencia de tiempos que tiene BT frente a FB con n no tan grande, y no tiene que ver tanto con el dataset usado ya que con el de densidad baja ocurriría lo mismo, debido a que la densidad no afecta demasiado el rendimiento de BT (cosa que otras instancias si logran como se observará en otro experimento). Y lo que verdaderamente hace la diferencia es la presencia de las podas en BT ya que reduce considerablemente los cálculos realizados.

Para enfatizar la ventaja que presenta usar un algoritmo con podas, en la Figura 2b se muestra el mismo gráfico con un n más chico donde se ve el gran salto de tiempo que tiene FB con respecto a BT desde el principio del experimento. Por otro lado se deja ver el ligero crecimiento de BT a mayor n ya que por más apañado que se vea BT en la Figura 2a no significa que no aumente a medida que crece n.

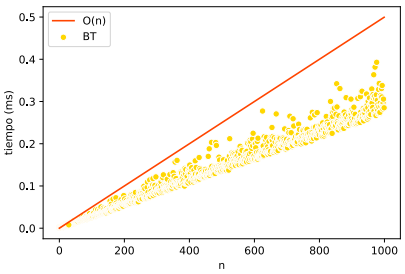
Por último, antes de hablar del siguiente gráfico cabe destacar que en los dos anteriores se pudo ver lo solapado que están los valores de tiempo de BF con respecto a la complejidad esperada, y justamente introducen al objetivo de la Figura 2c, donde se muestra el gráfico de correlación entre lo anteriormente mencionado para todas las instancias evaluadas, por lo que no es sorpresa saber que el índice de correlación de Pearson de las dos variables es de aproximadamente 0,9997667.

5.4. Experimento 2: Caso Lineal de Backtracking vs Caso Exponencial

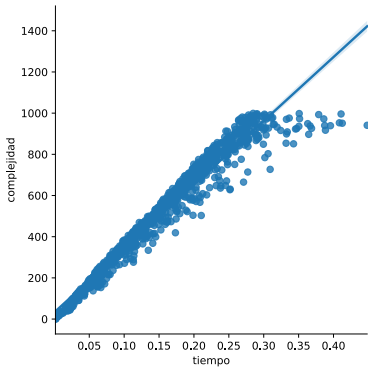
Durante la sección 3 se expuso acerca de los distintos sets de instancias que se podían analizar para el algoritmo. En este experimento se hablará de un set de instancias donde la complejidad del algoritmo se comportará de manera lineal, donde este será el mejor caso del algoritmo. Y se presentará otro set el cual está caracterizado por atacar en demasía el rendimiento de las podas y que por su complejidad se llegará a comportar como $O(2^{\frac{n}{2}})$. Cada uno de estos experimentos podrán ser visualizados en la Figura 3.

Primero se verá el comportamiento del primer set mencionado, llamado "mejor-caso-bt", y lo que tiene de especial es que solo el último elemento será solución, por lo que la complejidad de este algoritmo se comportará de manera lineal con respecto a la cantidad de elementos de entrada como se puede ver en la Figura 3a, ya que no tendrá que hacer una evaluación completa de cada combinación posible del conjunto de entrada, y solo se quedará con el último elemento como solución final, es decir, se comportará como $O(n)$. Cabe destacar un detalle de la figura donde se ve que si bien se comporta como una función lineal asintóticamente, se tiene una menor constante en la fórmula del tiempo del algoritmo. El comportamiento lineal de los tiempos se puede verificar en la Figura 3b, el cual es el gráfico de correlación del algoritmo con los tiempos esperados, donde se obtiene un índice de correlación de Pearson de aproximadamente 0,9952626.

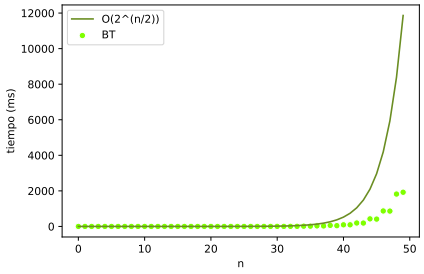
Por otro lado en la Figura 3c se puede ver como afecta el set de instancias "mal-caso-bt" al



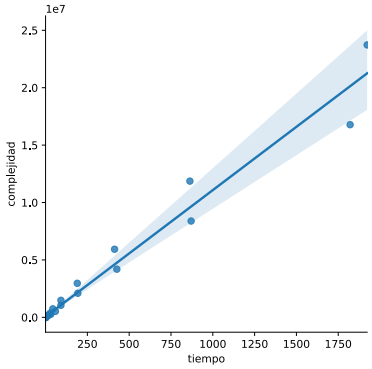
(a) Complejidad esperada vs Tiempo de ejecución en mejor caso de BT



(b) Correlación entre el tiempo de ejecución y la complejidad esperada del mejor caso



(c) Complejidad esperada vs Tiempo de ejecución en el caso malo de BT



(d) Correlación entre el tiempo de ejecución y la complejidad esperada del caso malo

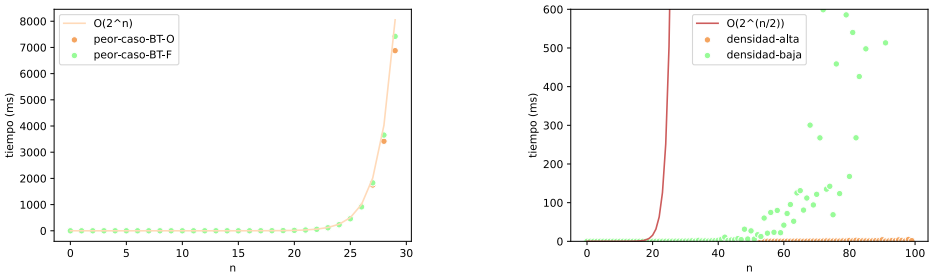
Figura 3: Análisis del método Backtracking en su mejor caso y en un mal caso.

rendimiento del algoritmo ya que a partir de cierto n_0 se exhibe su comportamiento exponencial, esto se produce debido a que las instancias están armadas de modo tal que las podas sean evitadas y empiece a comportarse como si fuera FB. La complejidad $O(2^{\frac{n}{2}})$ es lograda porque la mitad de los elementos de la lista de entrada va a ser solución del problema, mientras que la otra mitad será descartada de la solución final. Y esta complejidad es una "*mala complejidad*" (es decir no polinomial) demostrando así la ineficiencia de BT en algunos sets, ya que por más que se haya llegado a comportar de forma lineal en su mejor caso, el hecho de que sea exponencial en este caso evidencia su raíz, ya que se sabe que de fondo es un algoritmo de FB sumado a las podas que elegimos.

Por último en la Figura 3d, se deja ver la correlación con los tiempos que se esperaban, y se tiene que el índice de correlación de Pearson resulta en aproximadamente 0,9415130.

5.5. Experimento 3: Peor caso para cada poda de Backtracking

Siguiendo la corriente del experimento 2, este experimento trata de evidenciar los puntos débiles de BT con respecto a sus podas por separado, es decir BF-O y BF-F, comparando con la complejidad del algoritmo de FB, todo esto se encuentra plasmado en la figura 4. En la figura 4a se puede apreciar que en el peor caso de cada poda se comportará como una función exponencial, para ser más preciso como $O(2^n)$. Esto sucede debido a las instancias que se utilizaron para cada caso; para optimalidad se usó un conjunto de entrada el cual ninguno va a entrar en el tubo, y para factibilidad todo los elementos del conjunto de entrada serán parte del conjunto solución.



(a) Tiempo de ejecución de ambas podas de BT frente a $O(2^n)$

(b) Comparación de tiempos de BT con densidad alta y baja

Figura 4: Análisis de las podas de BT

Esto hace que las podas se anulen en cada caso respectivamente y logren que la complejidad del algoritmo sea la misma que la de FB.

Por otro lado siguiendo con el análisis de tiempos de BT, en la Figura 4b se vuelven a analizar las podas, pero esta vez juntas para poder visualizar que ante otros datasets los tiempos se ven muy distintos de lo que se esperaría, este es el caso de los datasets de densidad alta y baja, donde por el gráfico se puede ver que en densidad alta de datos, BT maneja bastante bien los tiempos, mientras que en densidad baja se tiene una suba exponencial de los tiempos a mayor n. Esto deja ver que BT funciona mejor para las instancias donde no hay demasiados elementos para colocar en el tubo, y análogamente para una densidad baja de elementos, es decir, cuando se tiene que guardar una gran cantidad de elementos, los tiempos de cómputo empiezan a subir exponencialmente, por lo que no sería conveniente usar BT para instancias de este estilo.

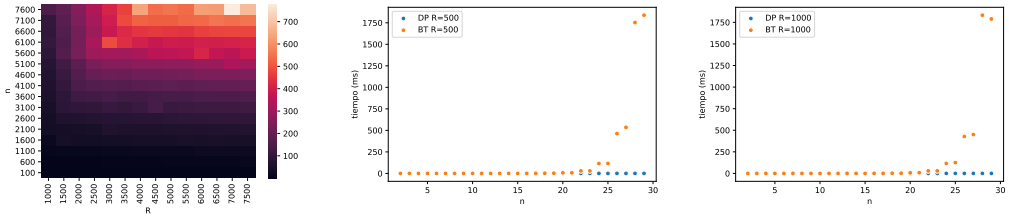
5.6. Experimento 4: Programación Dinámica vs Backtracking

En este último experimento se busca analizar a mayor profundidad la solución al problema de los jambo-tubos y su aplicación en el mundo real.

Anteriormente, se evidenció que BT tiene un mejor rendimiento que FB en el caso promedio, sin embargo en el caso de comparar BT vs DP no es tan fácil, ya que la complejidad del segundo depende de 2 variables y no es posible establecer una comparación directa. Sin embargo, se llevo el experimento a un posible caso de la vida real para determinar según esta posible instancia que algoritmo tiene un mejor comportamiento.

Primero es necesario comprender mas a fondo el comportamiento de esta implementación de DP. En la la Figura 5a se evidencia que DP tiene un buen rendimiento para un R relativamente chico e instancias de n grandes, lo cual en el caso de las otras dos técnicas tendría un costo mucho mas alto debido a que son exponenciales con respecto a n. Por otro lado se puede apreciar en el heatmap que a medida que se incremente R a la par de n aumenta el tiempo considerablemente, y seguramente para un R muy grande se tendrá que $n * R > 2^n$. Sin embargo, en la practica este caso no es común pues no existen jambo-tubos con resistencias casi infinitas. Por otro parte se presume que en la vida real la resistencia de un elemento va a ser proporcional a su peso, y aun mas en el caso del problema planteado relativo a los supermercados. Por ultimo vale destacar del heatmap que una solución al costo de usar instancias con R grande en la vida real es inicializar la matriz en base al $min(R, max(r_i))$, es decir tomando la resistencia mas grande del set de ítems a empacar, disminuyendo así la complejidad esperada y la memoria utilizada.

Finalmente, para el caso comparativo entre BT vs DP se plantea una familia de instancias de tamaño n desde 2 hasta 30 (para disminuir el tiempo de computo) e ítems de supermercado que pesen entre 1 y 10 kilos, con una resistencia de entre 100 y 200 kilos (considerando objetos como una bolsa de arroz por ejemplo) para ser empacados en un jambo-tubo con R de 500 y 1000 kilos. Al correr los experimentos para ambos R's se evidencio que en el caso promedio en la practica



(a) Tiempo de ejecución en funcion de n y R

(b) Backtracking vs DP con $R = 500$

(c) Backtracking vs DP con $R = 500$

Figura 5: Comparación de Backtracking frente a Programación Dinámica

el algoritmo de programación dinámica tiene un mejor rendimiento para el caso planteado, sin embargo vale la pena acotar que esto no implica que un algoritmo sea mas eficiente que otro mas a allá de una instancia especifica y que su elección de uso siempre debe ser circunstancial al problema en cuestión.

6. Conclusiones

A lo largo de la presente investigación se presentaron y compararon tres técnicas algorítmicas distintas para resolver el problema de optimización combinatoria de los Jambo-Tubos, empezando por Fuerza Bruta que tiene la implementación mas *naive* que se podría hacer de forma recursiva, donde se observa lo ineficiente que es a la hora de operar con grandes conjuntos de entrada. Luego, se presenta la técnica de Backtracking siendo ésta una mejora del algoritmo de FB gracias a las podas por factibilidad y optimalidad, al punto de que para ciertas familias de instancias su eficiencia alcanza tiempos que crecen linealmente sin incurrir en un mayor uso de memoria de computo, por ende siempre que el tamaño del código (en lineas o bytes) a ejecutar no sea una limitación (lo cual con los tamaños de de memoria actual seria poco plausible) es recomendado utilizar backtracking, también resulta interesante destacar que para n muy chicos la diferencia entre BT y FB es menos acentuada. Por último se expuso una solución no exponencial con respecto a n utilizando Programación Dinámica con una implementación *Top-Down* recursiva en donde se introdujo una estructura de Memoización que permite guardar todos los cálculos que se va computando para luego accederlos rápidamente cuando se los vuelva a necesitar, y así lograr un mejor rendimiento que las implementaciones anteriores, al coste de agregar un nuevo parámetro a la función de complejidad temporal, lo cual implica que para un R muy grande podría afectar el rendimiento pero en especial el uso la memoria. En sistemas donde la memoria no sea una limitación se recomienda utilizar DP para R's que no sean demasiado grandes.

Algunas mejoras que no se realizaron y que podrían abordarse en un futuro son la implementación de nuevas podas que reduzcan mas el árbol de recurrencia que las que se realizaron en la sección 3, estas se eligieron por no empeorar la complejidad teorica con respecto a FB pero para otros tipos de problema podas mas costosas pero inteligentes son utilizadas. Como también se podría optimizar al algoritmo de PD implementando una versión iterativa *Bottom-Up* que reduzca los llamados recursivos y el uso del stack o usando una estructura mas amigable con la memoria. Otra alternativa como se menciono en el Experimento 4 es elegir de forma inteligente el mejor valor de R antes de inicializar la matriz. Por ultimo también se podría haber realizado mas experimentos probando distintas instancias entre BT y PD para ver si en alguna BT tenia un mejor desempeño que PD en cuanto al tiempo de ejecución lo cual es plausible para un R grande.