DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Jambo Tubos

Algoritmos y estructuras de datos III

Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Leandro Rodriguez	521/17	leandro21890000@gmail.com
Andres Mauro	39/17	sebaastian_mauro@live.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Resumen

En el presente trabajo abordaremos el problema de los Jambo-tubos, resolviéndolo mediante la utilización de ciertas técnicas algorítmicas aprendidas en la materia, y luego realizando un análisis sobre la relación entre la complejidad obtenida experimentalmente y la complejidad teórica de los mismos.

Índice

1. Introducción

El problema de la mochila(Knapsack problem[1]) es uno de los problemas fundamentales de Ciencias de la Computación. En el presente trabajo resolveremos una variante del mismo, denominado problema de los Jambo-Tubos, que podemos formalizar de la siguiente manera: Dado un Jambo-Tubo con resistencia R, y una secuencia ordenada de n productos S, cada uno con un peso asociado w_i y una resistencia asociada r_i, determinar la máxima cantidad de productos que pueden apilarse (respetando el orden de S) en un tubo sin que ninguno quede aplastado.

Para simplificar las explicaciones, consideraremos solución a toda subsecuencia S' de S con $n' \leq n$ elementos y decimos que es factible si sus elementos no quedan aplastados en el tubo, ni lo rompen(es decir, que la resistencia del tubo R sea mayor o igual que $\sum_{i=0}^{n'-1} w_i$ y $(\forall j : \mathbb{N})(0 \leq j < n' \Longrightarrow \sum_{i=j+1}^{n'-1} w_i \leq r_j)$.) y no se puede agregar ningún otro elemento $w_i \in \{w_i \mid (w_i \in \text{S}) \wedge (w_i \notin \text{S'})\}$. En este trabajo resolveremos la variante de optimización, que indica cuál es el máximo cardinal de una subsecuencia factible S'. Para simplificar, decimos que si no existe ninguna subsecuencia factible, la respuesta es 0.

A continuación, se exhiben algunos ejemplos con sus correspondientes respuestas esperadas. Si la resistencia del tubo es 30, y S = {(10,45),(20,8),(30,15)} contiene respectivamente los w_i y r_i del elemento i, entonces existen 4 soluciones factibles S'_1={(10,45)}, S'_2={(20,8)}, S'_3={(30,15)}, S'_4={(10,45),(20,8)} y la de mayor cardinalidad es S'_4, por lo tanto la respuesta es 2. Por otra parte, si la resistencia del tubo es 5, y S = {(10,45),(20,8),(30,15)} entonces no existe ninguna solución factible, dado que cualquier elemento rompe el peso que aguanta el tubo. Por lo tanto, la respuesta en este caso es 0.

El objetivo de este trabajo es abordar el problema de los Jambo-Tubos utilizando tres técnicas de programación distintas y evaluar la efectividad de cada una de ellas para diferentes conjuntos de instancias. En primer lugar se utiliza Fuerza Bruta que consiste en enumerar todas las posibles soluciones, de manera recursiva, buscando aquellas que son factibles. Luego, se introducen podas para reducir el número de nodos de este árbol recursivo en busca de un algoritmo más eficiente, obteniendo un algoritmo de Backtracking. Finalmente, se introduce la técnica de memoización para evitar repetir cálculos de subproblemas. Esta última técnica es conocida como Programación Dinámica (DP, por sus siglas en inglés).

2. Fuerza Bruta

Un algoritmo de Fuerza Bruta enumera todo el conjunto de soluciones, en búsqueda de aquellas factibles u óptimas según si el problema es de decisión u optimización. En este caso, el conjunto de soluciones está compuesto por todos los conjuntos S' $\subseteq$ S que puedo formar, es decir todos los subconjuntos posibles de S. Observar que la cantidad de subconjuntos posibles es el conjunto de partes de S, que se escribe $\rho(S)$. Por ejemplo si R = 25 y S = {(10,45),(20,8)}, el conjunto $\rho(S)$ = {$\emptyset$,{(10,45)},{(20,8)},{(10,45),(20,8)}}, y el conjunto de soluciones factibles es {{(10,45)},{(20,8)}}.

La idea del algoritmo 1 para resolver el JTP(Jambo-Tubos Problem) es ir generando soluciones de manera recursiva decidiendo en cada paso si considerar o no un elemento de S y quedándose con la mejor solución de alguna de las dos ramificaciones. En nuestro caso, la mejor solución dadas dos ramificaciones diferentes del algoritmo, será el cardinal de aquella factible que maximice la cantidad de elementos posibles.

En la figura 1 se ve un ejemplo del árbol de recursión para la instancia S = {(10,45),(20,8),(30,15)} y R = 30. Cada nodo intermedio del árbol representa una solución parcial, es decir, cuando aún no se tomaron todas las decisiones de qué elementos incluir, mientras que las hojas representan a todas las soluciones (8 en este caso). La solución óptima (10,45),(20,8) está marcada en rojo y las otras soluciones factibles se encuentran marcadas en gris. Notar que la solución al problema original es BruteForce(S, 0, R, 0).

La correctitud del algoritmo se basa en el hecho de que se generan todas las posibles soluciones, dado que para cada elemento de S se crean dos ramas una considerándolo en el conjunto y la otra en el caso

[1] https://en.wikipedia.org/wiki/Knapsack_problem

Algorithm 1 Algoritmo de fuerza bruta para JTP

```
1:  function BruteForce(Jmbo_e, i, R, cant)
2:      if i = Jmbo_e.Size() then
3:          if R >= 0 then return cant;
4:          else
5:              return 0;
6:          end if
7:      else
8:          noAgregar ← BruteForce(Jmbo_e, i+1, R, cant);
9:          Agregar ← BruteForce(Jmbo_e, i+1, min(Jmbo_e[i].res, R - Jmbo_e[i].peso), cant++);
10:         return max(noAgregar,Agregar);
11:     end if
12: end function
```

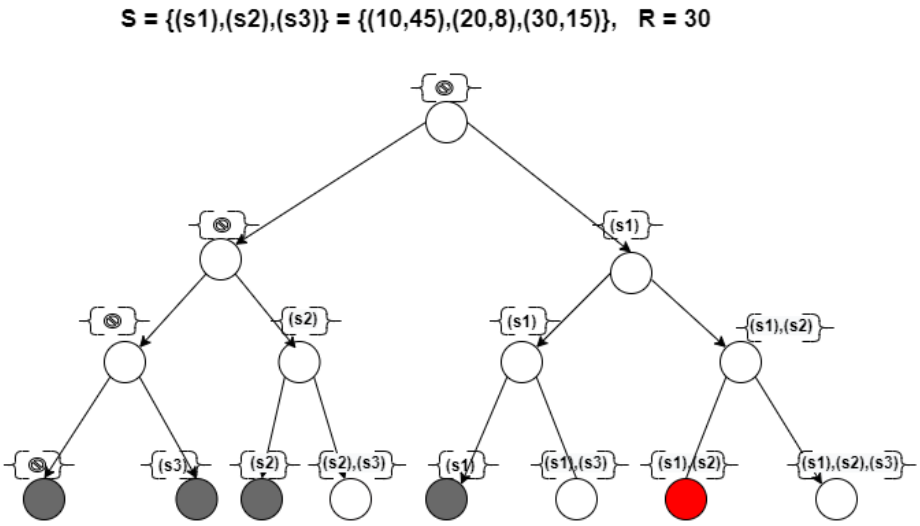


Figura 1: Ejemplo de ejecución del Algoritmo 1 para S = {(10,45),(20,8),(30,15)} y R = 30. En gris las soluciones factibles y en rojo la solución óptima.

contrario. Al haber generado entonces todas las posibilidades, la óptima debe encontrarse en caso de que esta exista. La complejidad del Algoritmo 1 para el peor y mejor caso es $\Theta(2^n)$.

Esto se debe a que independientemente del valor de las instancias, el árbol de recursión que se genera, es un árbol binario completo de n+1 niveles(contando la raíz), dado que cada nodo se ramifica en dos hijos y en cada paso el parámetro i es incrementado en 1 hasta llegar a n. Además, podemos observar que tanto el caso base como los llamados recursivos se resuelven en tiempo constante(lineas 3-6 y 8-10 respectivamente), ya que constan de operaciones elementales como suma, mínimos y comparaciones.

3. Backtracking

El algoritmo de backtracking se relaciona fuertemente con el de fuerza bruta, su diferencia recae en que antes de continuar extendiendo una solución parcial considera si esto es o no viable. Esto le permite disminuir la cantidad de soluciones candidatas a explorar.

Decimos que una solución parcial es una instancia viable, si es posible extenderla a una solución óptima. Este algoritmo, dado una instancia viable genera un árbol de backtracking, tomando todas las posibles decisiones locales; cada nodo o rama del árbol, contiene una solución parcial y sobre esta, se aplican criterios para decidir si se debe seguir explorando esta rama para encontrar la solución óptima.

Esta serie de criterios son conocidos como podas, y aunque cada poda depende del problema en particular a resolver, estas se dividen en dos categorías: *factibilidad* y *optimalidad*.

3.1. Poda por factibilidad

En este caso, una poda por factibilidad es la siguiente. Sea S_0 una solución parcial representada con un nodo intermedio n_0 con R < 0. Claramente, al ser R < 0 no va a haber ninguna forma de extender S_0 de manera tal que no se haya roto ningún elemento ni el tubo. De este modo, podemos evitar seguir explorando el subarbol formado debajo de n_0 y por lo tanto, reducir la cantidad de operaciones de nuestro algoritmo. Esta poda está expresada en las lineas 2-3 del Algoritmo 2.

3.2. Poda por optimalidad

Supongamos que ya se conoce una solución factible para el problema, con cantidad de elementos K_0. Además, supongamos que se esta en un nodo intermedio n_0 que representa a una solución parcial S_0. S_0 tiene K_1 elementos agregados al tubo y J elementos por delante a analizar si incluir en la solución o no. En este caso, si $K_1 + J < K_0$, como cualquier solución parcial a la que se llegue partiendo desde este nodo va a tener menos elementos agregados que la solución ya encontrada, podemos asegurar que cualquier solución factible que se encuentre al explorar no va a ser óptima.

Por lo tanto, se puede podar esta rama y así evitar el cómputo innecesario de operaciones. En el Algoritmo 2 se actualiza una variable global cantMax cada vez que se halla una solución factible (mejor a la anterior encontrada) en las lineas 8-9, y se evalúa la regla de la poda en la línea 11.

Algorithm 2 Algoritmo de Backtracking para JTP

```
1:  function Backtracking(Jmbo_e, i, R, cant)
2:      if R < 0 then
3:          return −∞                                   ▷ poda por factibilidad
4:      end if
5:      if i = jmbo_e.size() then
6:          return 0                                    ▷ caso base
7:      end if
8:      if cant > cantMax then
9:          cantMax ← cant                              ▷ actualizo el maximo
10:     end if
11:     if (cant + (jmbo_e.size() − i)) < cantMax then  ▷ poda por optimalidad
12:         return −∞
13:     end if
14:     noAgregar ← BackTracking(jmbo_e, i+1, R, cant)
15:     Agregar ← BackTracking(jmbo_e, i+1, min(jmbo_e[i].res, R − jmbo_e[i].peso), cant+1)
16:     return max(noAgregar, Agregar)
17: end function
```

3.3. Complejidad

La complejidad del algoritmo en el peor caso es $O(2^n)$. Esto es así, porque en el peor escenario no se logra podar ninguna rama y por lo tanto se termina enumerando el árbol completo al igual que en Fuerza Bruta. Además, se puede observar que el código introducido en el caso base y los llamados recursivos solamente agrega un número constante de operaciones.
Existe una familia de instancias para las cuales este algoritmo no va a realizar ninguna poda por factibilidad, estas son las instancias en donde la suma de todos los pesos de los elementos no supera la resistencia del tubo ni la resistencia de ningún elemento, explicado más precisamente, son aquellos S tales que:

$$\sum_{i=0}^{|S|-1} w_i \leq R \wedge (\forall j : \mathbb{N})(0 \leq j < |S| \Longrightarrow \sum_{i=j+1}^{|S|-1} w_i \leq r_j).$$

Por ende en la solución óptima, quedan agregados todos los elementos de la cinta al jambo-tubo.
Por otro lado, la poda por optimalidad nunca se realiza cuando no es posible agregar ningún elemento de la cinta al jambo tubo, por ejemplo cuando:

$$(\forall j : \mathbb{N})(0 \leq j < |S| \implies w_j > R),$$

ya que de esa manera la solución óptima quedaría con 0 elementos y nunca se cumpliría la guarda de la linea 11 del algoritmo 2 y por ende nunca poda.
Sin embargo, durante la investigación no se encontraron instancias que anulen en su totalidad ambas podas simultáneamente y obliguen al algoritmo a recorrer todos los nodos. Esto se debe a que si dada una instancia, la solución óptima del problema es no agregar ningún elemento (para no realizar podas por optimalidad), cada elemento que se agregue provocaría una poda por factibilidad (ya que se rompería el tubo). Y dada una instancia en la que todos los elementos se pueden agregar (para no podar por factibilidad), se efectuarían podas por optimalidad en ramas en las cuales no agreguen todos los elementos. Esto es un indicio de que la complejidad del algoritmo puede ser menor a la cota provista ($O(2^n)$). Sin embargo, más adelante en la experimentación vemos 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 2 instancias genéricas: cuando en la solución no se ingresa ningún elemento (dado que tienen un peso mayor a la resistencia del jambo-tubo) y cuando la solución es un único ítem (el último de la cinta) para agregar (pues todos los demás tienen un peso mayor a la resistencia del jambo-tubo). Como el primer llamado recursivo siempre no agrega un elemento, la mejor solución es la primera (en el primer caso) y la segunda (en el segundo caso) en encontrarse, en consecuencia las ramificaciones de todos los demás nodos no se exploran.

4. Programación dinamica

Los algoritmos de Programación Dinámica entran en juego cuando una función recursiva tiene superposición de subproblemas. La idea consiste en evitar recalcular todo el subárbol correspondiente si ya fue calcuado con anterioridad.
En nuestro caso proponemos la siguiente función recursiva que resuelve el problema:

$$PD(i,R) = \begin{cases} -\infty & si \quad R < 0 \\ 0 & si \quad i = n \\ max(PD(i+1,R), PD(i+1, min(R - w_i, r_i)) + 1) & cc \end{cases} \quad (1)$$

La ecuación coloquialmente está diciendo que la máxima cantidad de elementos $S_i, ..., S_{n-1}$ que se puede poner (respetando su orden) en el tubo cuya resistencia es R, es igual a 0 si ya no quedan más elementos que agregar ($i = n$), $-\infty$ si ya se rompió el tubo o se aplasto alguno de los elementos($R < 0$) o caso contrario, el máximo valor resultante, entre no agregar el i-esimo elemento, o agregarlo sumando uno a la cantidad total de elementos en el tubo.
Al agregarlo, se debe recalcular la resistencia actual (quedándose con la mínima resistencia entre, la resistencia del tubo menos el peso del i-esimo elemento o la resistencia del i-esimo elemento).
Cabe aclarar que el llamado a la función que resuelve el problema entero es: $PD(0, R)$ con R la resistencia original del tubo.

4.1. Correctitud

1. Si $R < 0$ significa que el tubo o alguno de los elementos en su interior superó su resistencia máxima y colapsó, por lo que no es una solución factible, entonces la respuesta es $PD(i, R) = -\infty$.

2. Si $i = n$ significa que no hay más elementos para colocar, por lo que la respuesta es $PD(i, R) = 0$.

3. Ya que ahora $i < n$ y $R \geq 0$ se necesita encontrar la forma de retornar la máxima cantidad tal que se cumpla la premisa. Separemos el análisis en dos ramas, si se coloca el ıtem i en el tubo o si no. Si no se coloca se debe seguir analizando que pasa al agregar el siguiente ıtem, es decir que se calcula $PD(i+1, R)$. Ahora la parte interesante pasa si se incluye el ıtem 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 ıtem i ya agregó un peso extra al tubo, la nueva resistencia será el mínimo entre la resistencia del ítem i y la vieja resistencia del tubo menos el peso de i, es decir el $min(R-w_i, r_i)$. Ası que se puede concluir que este caso es igual a $PD(i+1, min(R-w_i, r_i))+1$. Dicho todo esto, la respuesta será el máximo entre los resultados de ambas ramas, o sea: $max(PD(i+1, R), PD(i+1, min(R-w_i, r_i))+1)$

4.2. Memoización

Notemos que la función recursiva (1) toma dos parámetros i $\in 1, ..., n-1$ y r $\in 0, ..., R$. Notar que los casos $i = n$ o $R < 0$ son casos base y se pueden resolver de manera ad-hoc en tiempo constante. Por lo tanto, la cantidad de posibles estados con la que se puede llamar a la función, o combinación de parámetros, está determinada por la combinación de ellos. En este caso, hay $O(n * R)$ combinaciones posibles de parámetros. Mientras que la cantidad de llamados recursivos que se realizan es $O(2^n)$. Lo cual indica que hay superposición de subproblemas, en base a esto, parecería una buena idea implementar un algoritmo de programacion dinámica para resolver el problema. En este sentido, si agregamos una memoria que recuerde cuando un caso ya fue resuelto y su correspondiente resultado, podemos calcular una sola vez cada uno de los problemas y asegurarnos no resolver más de $O(n * R)$ casos. El Algoritmo 3 muestra esta idea aplicada a la función recursiva (1). En la línea 8 se lleva a cabo el paso de memoización que solamente se ejecuta si el estado no había sido previamente computado.
Aclaración: M es la estructura de memoizacion, una matriz de n filas y R columnas tal que

$$M[k][h] = \begin{cases} \perp & si \quad PD(k,h)\ no\ fue\ calculado\ con\ anterioridad \\ PD(k,h) & si \quad PD(k,h)\ fue\ calculado\ con\ anterioridad \end{cases}$$

para ello, M es inicializada con $\perp$ en cada celda antes de ejecutar el algoritmo

Algorithm 3 Algoritmo de DP para JTP

```
1:  function DP(Jmbo_e, i, R)
2:      if R < 0 then
3:          return −∞
4:      end if
5:      if i = jmbo_e.size() then
6:          return 0
7:      end if
8:      if M[i][R] = ⊥ then
9:          NoAgregar ← PD(jambo_elementos, indice+1 ,Resistencia)
10:         Agregar ← PD(jmbo_e,i+1,min(jmbo_e[i].res,R−jmbo_e[i].peso))
11:         M[i][R] ← max(noAgregar, Agregar)
12:     end if
13:     return M[i][R]
```

4.3. Complejidad

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 3 realizan operaciones constantes, cada estado se

resuelve en $O(1)$. Como resultado, el algoritmo tiene complejidad teórica $O(n * R)$ en el peor caso. Notar que la inicialización de M 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 presentaremos los experimentos computacionales realizados para contrastar el análisis teórico presentado anteriormente con resultados prácticos en una máquina física. En particular, los experimentos fueron llevados a cabo en una workstation con CPU Intel Core I5 @ 3.1 GHz y 8GB de memoria RAM, y utilizando el lenguaje de programación C++.

5.1. Métodos

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

- **BF:** Algoritmo 1 de Fuerza Bruta de la sección 2.
- **BT:** Algoritmo 2 de Backtracking de la sección 3.
- **BT-F:** Algoritmo 2 solo aplicando podas por factibilidad, es decir descartando las lineas 8-13.
- **BT-O:** Algoritmo 2 solo aplicando podas por optimalidad, es decir descartando las lineas 2-4.
- **DP:** Algoritmo 3 de Programación Dinámica de la Sección 4.

5.2. Instancias

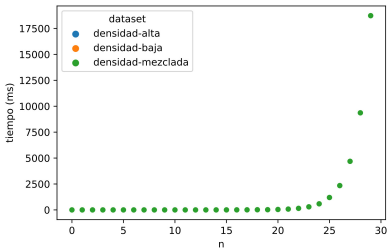
Para evaluar los algoritmos en distintos escenarios es preciso definir familias de instancias conformadas con distintas características. Por ejemplo, el algoritmo de Backtracking como se menciona en la Sección 3 tiene familias que producen mejores y peores casos para el algoritmo. Primero, antes de enumerar los datasets, definiremos la densidad de una instancia como $\frac{R+\sum_{i=1}^{n} r_i}{\sum_{i=1}^{n} w_i}$, es decir, es una medida de cuántos elementos de la cinta transportadora acabaran finalmente en el tubo. A mayor densidad, los pesos de los elementos de la cinta son chicos en relación a sus respectivas resistencias (incluyedo la resistencia del tubo) y por lo tanto será posible apilar más de ellos. Finalmente, los datasets definidos se enumeran a continuación.

- **densidad-alta:** En esta familia, los elementos de cada instancia tienen peso entre $1, ..., n$ y la resistencia de cada uno, es dos veces su peso. Están ordenados de manera aleatoria. Y la resistencia del jambo tubo es $2 * n$.
- **densidad-media:** En esta familia, los elementos de cada instancia tienen peso entre $1, ..., n$ y la resistencia de cada uno, es igual a su peso. Están ordenados de manera aleatoria. Y la resistencia del jambo tubo es $\frac{n*(n+1)}{2}$ (esta instancia solo se utiliza para generar las instancias de densidad mezcla).
- **densidad-baja:** En esta familia, los elementos de cada instancia tienen peso entre $1, ..., n$ y la resistencia de cada uno, es la mitad de su peso. Están ordenados de manera aleatoria. Y la resistencia del jambo tubo es $\frac{n}{2}$.
- **densidad-mezcla:** En esta familia, los elementos de cada instancia tienen un tercio de cada una de las anteriores instancias. Están ordenados de manera aleatoria. Y la resistencia del jambo tubo es $\frac{R_1+R_2+R_3}{3}$ (siendo que estos R_i, son el valor de la resistencia del tubo en cada una de las instancias).
- **bt-mejor-caso :** Cada instancia de n elementos, está formada por elementos que su peso es mayor a la resistencia del tubo.

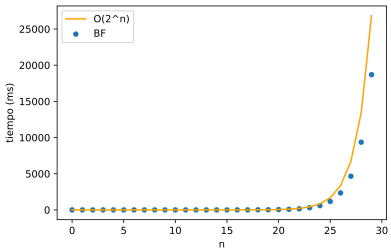
- **bt-mal-caso:** Cada instancia de n elementos (n par) en su primera mitad tiene $\frac{n}{2}$ elementos que conforman la solución de su problema, es decir cuya suma de pesos es menor a la resistencia del tubo, y en la otra mitad no va a tener ningún elemento que deba pertenecer al tubo pues su peso es mayor a la resistencia del tubo. Este caso particular minimiza las podas y genera una complejidad exponencial.
- **Dinámica:** Esta familia de instancias se conforma de aquellas con distintas combinaciones de valores para la cantidad de elementos en la cinta y la resistencia del tubo en los intervalos [1000, 8000]. Los elementos de la cinta pertenecen a la densidad mezcla, ya que es la más representativa.

5.3. Experimento 1: Complejidad de fuerza bruta

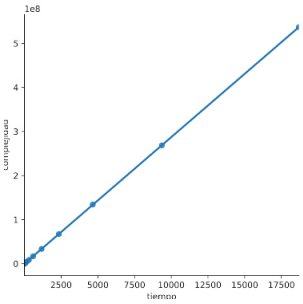
En este experimento se analiza la performance del método FB en distintos contextos. El análisis de complejidad teórico realizado en la Sección 2 indica que el tiempo de ejecución para el mejor y peor caso es idéntico y es exponencial en función de n. Para contrastar empíricamente estas afirmaciones se evalúa FB utilizando los datasets densidad-alta, densidad-mezcla y densidad-baja y se grafican los tiempos de ejecución en función de n. La Figura 2(a) presenta los resultados del experimento, donde se puede apreciar que las tres curvas están solapadas para todas las instancias. El mensaje principal de este gráfico es que los tiempos de ejecución parecen no alterarse según la densidad de las instancias y seguir la misma curva de crecimiento sin importar las características de las mismas. A continuación, tomamos la ejecución sobre el dataset densidad-alta y evaluamos cuál es su correlación con la complejidad estudiada en la Sección 2, es decir, $\Theta(2^n)$. En la Figura 2(b) se ilustra el tiempo de ejecución de FB a la par de una función exponencial de $\Theta(2^n)$. Por otro lado, para la Figura 2(c) graficamos los tiempos de ejecución de la serie de instancias $X_1, ..., X_n$ vs la complejidad esperada. y para cada una se grafica el tiempo de ejecución real $T(X_i)$ contra el tiempo esperado $E(X_i) = 2^i$, es decir, su gráfico de correlación. Se puede ver que el tiempo de ejecución sigue claramente una curva exponencial y además la correlación con la función 2^n es positiva y casi perfecta. En particular, el índice de correlación de Pearson de ambas variables es $r \cong 0{,}9999$. Por lo tanto, podemos afirmar que el algoritmo se comporta como se describió inicialmente en las hipótesis.



(a) Tiempo de ejecución BF densidades alta, baja y mezcla.



(b) Tiempo de ejecución BF contra complejidad esperada.

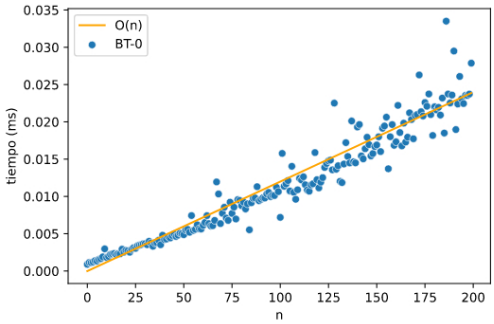


(c) Correlación entre el tiempo de ejecución y complejidad esperada.

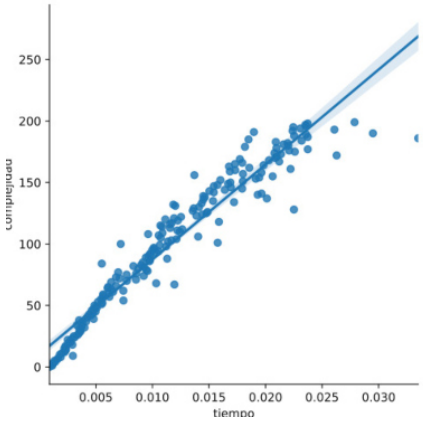
Figura 2: Análisis de complejidad de Fuerza Bruta para instancias alta, baja y mezcla.

5.4. Experimento 2: Complejidad de Backtracking

En esta experimentación vamos a contrastar las hipótesis de la Sección 3 con respecto a las familias de instancias de mejor y peor caso (encontrada) para el Algoritmo 2, y su respectiva complejidad. Para esto evaluamos el método BT con respecto los datasets bt-mejor-caso y bt-mal-caso. Las figuras 3 y 4 muestran los gráficos de tiempo de ejecución de BT y de correlación para cada dataset respectivamente. Efectivamente, las hipótesis presentadas anteriormente se cumplen para ambos casos. Por un lado, para las instancias de mejor caso se puede ver que efectivamente la serie de puntos muestra un crecimiento lineal aunque presenta cierto ruido. Uno de los motivos para este comportamiento es que al ser un comportamiento lineal, los tiempos de ejecución son muy bajos para incluso $n = 200$. Como resultado, cualquier interferencia en el sistema operativo o cambio de contexto puede causar una fluctuación indeseada y alterar los resultados. Sin embargo, el índice de correlación de Pearson es $r \cong 0{,}9636$ lo cuál muestra que hay una correlación positiva fuerte entre los tiempos de ejecución y una función lineal. Por otra parte, para las instancias de mal caso, los tiempos de ejecución se presentan más ajustados a la curva de complejidad exponencial $O(2^{\frac{n}{2}})$. Notemos que en este caso se ejecutaron instancias hasta n = 67, número elegido para evitar que el tiempo de ejecución sea demasiado grande ($>$ 10 minutos). Para estas instancias el índice de correlación de Pearson es de $r \cong 0{,}9177$ contra una función exponencial con base 2.

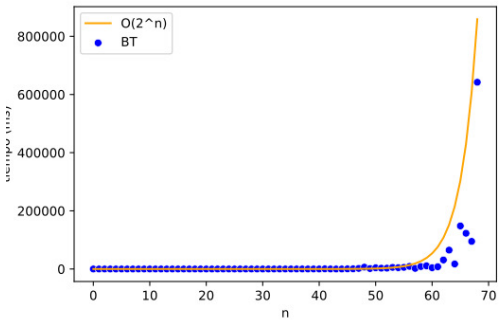


(a) Tiempo de ejecución BT mejor caso.

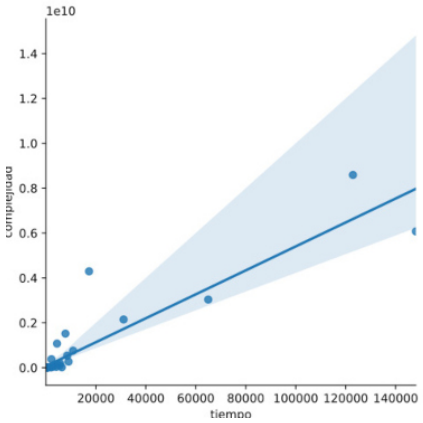


(b) Correlación BT mejor caso función lineal.

Figura 3: Análisis de complejidad para instancias de mejor caso BT.



(a) Tiempo de ejecución BT mal caso.



(b) Correlación BT mal caso función exponencial.

Figura 4: Análisis de complejidad para instancias de mal caso BT.

5.5. Experimento 3: Análisis de las podas

En este experimento queremos contrastar el análisis teórico realizado en la sección 3, sobre el comportamiento del algoritmo 2 y comparar experimentalmente si efectivamente los tiempos obtenidos al realizar las distintas podas (BT, BT-F, BT-O) dan el resultado esperado. Para realizar la experimentación, utilizaremos los dataset de densidad alta, baja y mezcla, interesandonos más bién en éste ultimo ya que se trata de una instancia que podemos suponer representativa de instancias que podríamos encontrar en casos de aplicación reales.

Asimismo, nuestra hipótesis antes de conducir el experimento consiste en que las instancias de menor densidad serán más eficientes que aquellas de mayor densidad para los casos en los que se pode por factibilidad, mientras que cuando se pode por optimalidad, las instancias más eficientes podrían ser las de mayor densidad. La justificación de esto es que cuando la densidad es menor, se minimiza la cantidad de elementos ingresados en el tubo, por lo cual habría más podas por factibilidad y menos por optimalidad.

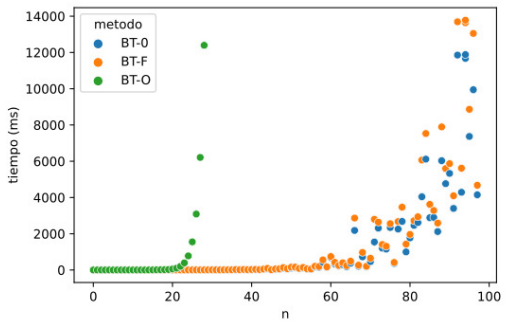
En la figura 5 podemos ver los gráficos de tiempo de ejecución según cada algoritmo en función de n para los distintos datasets de densidad alta, baja y mezcla. Notar que en la figura 5(c), obviamos el comportamiento de la poda de backtracking por optimalidad debido a que el resultado obtenido es similar a los anteriores, y queremos hacer énfasis, en que el resultado obtenido al realizar BT y BT-F es similar.

Efectivamente podemos notar, que para el dataset densidad baja, al aumentar las podas por factibilidad, nuestra hipótesis se corresponde con los resultados obtenidos. Siendo que en ese caso, las instancias de densidad baja hacen que aumente el tiempo de cómputo cuando solo se poda por optimalidad. De manera análoga, las instancias de densidad alta hacen que disminuya el tiempo de cómputo al podar por optimalidad. Se observa que cuando n es 25, para el dataset densidad baja el tiempo en milisegundos que tarda la ejecución del algoritmo es 5000, mientras que en el caso densidad alta el tiempo es 3000. Luego, se puede observar que no hay mucha variación en el tiempo de cómputo en los datasets para el caso en que podamos por factibilidad, de alguna manera contradiciendo nuestra hipótesis. Es decir, la eficacia de las podas realizadas por factibilidad no se ve muy afectada aún cuando el dataset es altamente denso. Este resultado nos dice, que aún cuando el dataset es muy denso, las podas por factibilidad siguen funcionando muy bien. Con lo cual, instancias que maximicen las podas por optimalidad van a ser poco probables de ser encontradas en casos reales. Por último, observamos que realizar las dos podas o podar solo por factibilidad no representa una diferencia significativa, hecho que se encuentra explicado justamente por lo mencionado anteriormente: para las instancias pertenecientes a los datasets de densidad alta, baja y mezcla, las podas por factibilidad están significando una muy buena reducción del tiempo de cómputo.

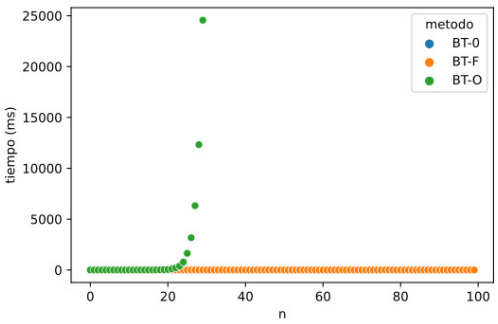
5.6. Experimento 4: Complejidad de programación dinámica

En este experimento contrastaremos la complejidad teórica del algoritmo de programación dinámica estudiada en la sección 4, con resultados experimentales obtenidos al ejecutar el algoritmo de DP sobre el dataset dinámica descripto en la sección 5.2.

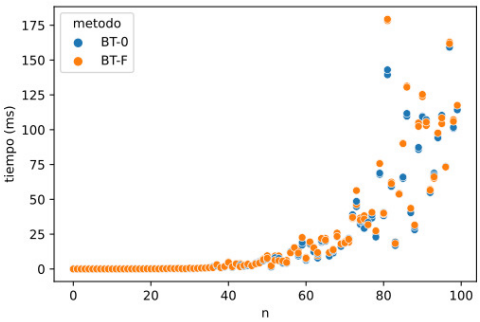
Las figuras 6(a) y 6(b) muestran el crecimiento del tiempo de ejecución en función de n y R respectivamente, sobre algunos cortes hechos en la otra variable. Se puede apreciar en ambos gráficos un crecimiento lineal en función de ambas variables. En la figura 6(c) se puede apreciar un gráfico de calor, donde las variables incidentes en el mismo son n y R, y en el cual podemos ver que el crecimiento de las mismas es similar. Por último, podemos ver en el gráfico 6(d) la correlación entre el tiempo de ejecución del algoritmo de DP sobre el dataset de densidad mezcla y el tiempo de ejecución teórico planteado en la sección 4. Podemos observar que efectivamente, hay una fuerte correlación positiva entre el tiempo obtenido experimentalmente al ejecutar el algoritmo de DP sobre un instancia representativa del mundo real, con el tiempo teórico propuesto en la sección 4 de $O(n^*R)$. En particular observamos que el índice de la correlación de Pearson resultante es $r \cong 0{,}9911$, con lo cual podemos decir fuertemente que experimentalmente los resultados fueron los esperados.



(a) Densidad alta.

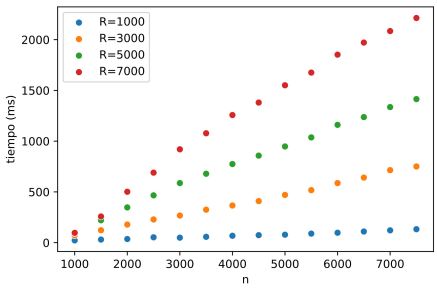


(b) Densidad baja.

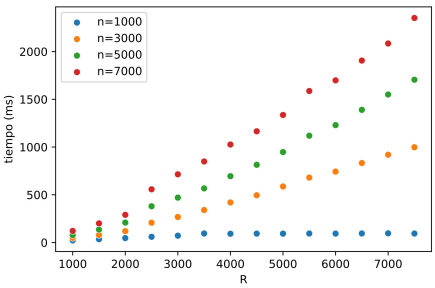


(c) Densidad mezcla.

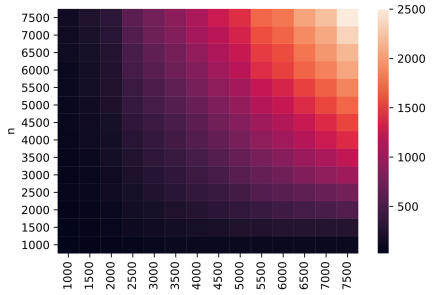
Figura 5: Análisis de BT, BT-F y BT-O para densidades alta, baja y mezcla.



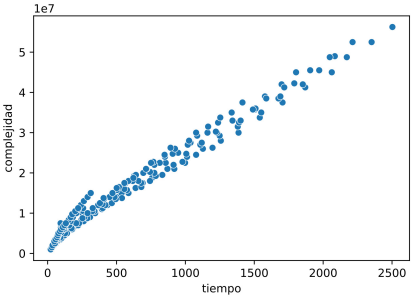
(a) Tiempo de ejecución en función de n.



(b) Tiempo de ejecución en función de R.



(c) Tiempo de ejecución en función de n y R.



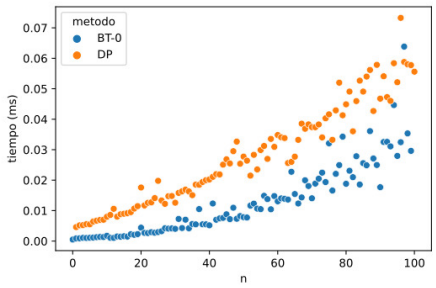
(d) Correlación entre el tiempo de ejecución y la cota de complejidad temporal.

Figura 6: Resultados computacionales para el método DP sobre el dataset dinamica.

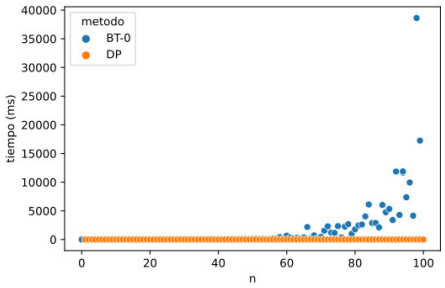
5.7. Experimento 5: Comparación Backtacking y Programación Dinámica

Para finalizar, presentamos un experimento que compara dos técnicas algorıtmicas distintas. La idea es obtener información que permita entender el comportamiento de cada método y que sirva para la toma

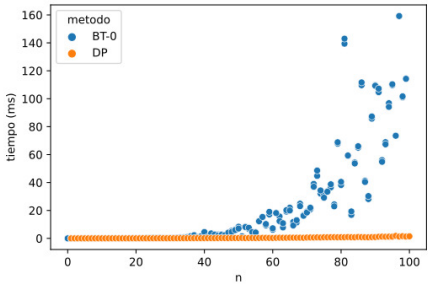
de decisión al momento de elegir alguno. Nuestra hipótesis es que ambos algoritmos van a comportarse mejor en situaciones distintas. Por ejemplo, Backtracking funciona muy bien en las instancias de densidad baja, y sus podas pueden llegar a ser muy efectivas en comparación con el alto costo de mantenimiento de la estructura de memoización de programación dinámica. Sin embargo, cuando la densidad es alta programación dinámica debe ser más eficiente. Para ello se evalúan y comparan ambos algoritmos con instancias de densidad baja, mezcla (que es la más representativa de un caso real) y alta. Una observación importante es que ningún algoritmo domina al otro en términos de complejidad. Dicho de otro modo, no es cierto que $O(2^n) \subseteq O(n * R)$ ni tampoco que $O(n * R) \subseteq O(2^n)$.



(a) Densidad baja.



(b) Densidad alta.



(c) Densidad mezcla.

Figura 7: Comparación de BT y PD para densidades alta, baja y mezcla.

La Figura 7 muestra la comparación entre los métodos DP y BT para los datasets densidad mezcla y densidad-alta. La hipótesis se confirma, mostrando que DP es más efectivo, ante instancias más cercanas a lo esperado en la vida real y instancias de alta densidad, que BT. Notar que los tiempos de ejecución son bajos en ambos tipos de instancias para DP. Sin embargo el crecimiento de BT en las instancias de densidad baja, es claramente mejor que el de PD aunque puede notarse que hay una clara relación lineal entre los tiempos de ejecución de los mismos. Esto hace que en este tipo de instancias la elección segura sea utilizar el algoritmo BT.

6. Conclusiones

En este trabajo presentamos una solución al problema de los Jambo-Tubos. Proponemos tres diferentes técnicas algorítmicas para abordar el problema, y explicamos las incumbencias de la utilización de cada uno de ellos. Vimos que con la técnica algorítmica de fuerza bruta podemos asegurarnos de hallar siempre una solución, pero los tiempos de ejecución del algoritmo aumentan exponencialmente a medida que crece el tamaño de la entrada. Por otro lado, la técnica algorítmica de backtracking presenta una mejora sustancial en tiempo de ejecución para resolver el problema, dicha mejora condicionada a la elección de diferentes podas que pueden realizarse. Presentamos un análisis teórico y su contrastación experimental para ciertas podas seleccionadas, aunque podrían pensarse otras dependiendo de los diferentes contextos de uso que se quieran dar. Por último, introducimos la técnica de programación dinámica, la cual ofrece un tiempo de cómputo más razonable que las otras para instancias representativas del mundo real.