DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP 1: Problema Jambo-tubos

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Sosa, Patricio	218/16	patriciososa91@gmail.com
Mamani, Carlos	496/16	mamanimezacarlos@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

Índice

1. Introducción

En el problema de los Jambos Tubos (PJT), consiste en optimizar la cantidad de productos que se pueden agregar en un tubo, donde cada producto posee un peso y resistencia asociada. La capacidad del tubo esta determinado por un valor y además se cuenta con la condición de que no superar la resistencia que posea cada producto. Formalmente dado un Jambotubo con resistencia $R \in \mathbb{N}$, y una secuencia ordenada de n productos S,tal que $n \geq 0$. Cada elemento s_i cuenta con un peso asociado w_i y una resistencia asociada r_i ambos enteros no negativos. El PJT consiste en determinar la mayor longitud de una subsecuencia $S' \subseteq S$, tal que:

- $\sum_{s_i \in S'} w_i \leq R$
- $(\forall s_i \in S')\ r_i \geq \sum_{\substack{s_k \in S' \\ \wedge k > i}} w_k.$

Cuando una subsecuencia S' cumple con las características mencionadas decimos que es una solución factible. Si no existe una ninguna subsecuencia factible se devuelve 0. Podemos pensar a S como un conjunto, pero respetando el orden en que aparecen los productos en la secuencia, en tal caso, el conjunto de soluciones candidatas es el *conjunto de partes de S* que se escribe $\mathcal{P}(S)$. A continuación se muestra el Ejemplo 1.1 de lo enunciado, con su correspondiente respuesta esperada.

Ejemplo 1.1. *Consideramos el conjunto* $S = \{s1, s2, s3\}$*, sus soluciones candidatas están dadas por el conjunto de partes* $\mathcal{P}(S) = \{\ \emptyset, \{s1\}, \{s2\}, \{s3\}, \{s1, s2\}, \{s1, s3\}, \{s2, s3\}, \{s1, s2, s3\}\ \}$*. Y además los siguientes datos:*

- $w = \{7, 4, 2\}$ *representa el peso de cada producto de* S*.*
- $r = \{5, 2, 1\}$ *representa la resistencia de cada producto* S*.*
- $R = 10$ *representa la resistencia del tubo.*

Las soluciones factibles son: $\{s1, s3\}$*,* $\{s2, s3\}$*, y en este caso la solución al problema es 2 por ser la máxima cantidad de productos que se pueden agregar.*

1.0.1. Máxima resistencia

Para que una solución candidata S' sea considerada factible, debe cumplir que la suma de los pesos en S' no supere el valor de R, y además que la suma de los pesos sobre el producto i, no supere el valor r_i, es decir la resistencia del producto i. Para respetar las restricciones que debe verificar una posible solución, se introduce una variable "máxima resistencia". Esta variable nos indica en todo momento cual es el peso máximo que se puede cargar en el tubo, de manera tal que no se supere la resistencia de ningún producto previamente elegido, ni se supere la capacidad R del tubo. Al inicio del problema, la máxima resistencia tiene el valor R, pues todavía no se agregó ningún producto. Al momento de decidir agregar o no el iesimo producto, sucede lo siguiente: Si el iesimo producto se agrega: el nuevo valor de máxima resistencia puede ser el valor r_i o puede ser la máxima resistencia que teníamos hasta el momento menos el peso del iesimo producto. Para conservar la semántica de la variable, la nueva máxima resistencia es el mínimo de los dos valores considerados.

En particular, si w_i supera el valor de máxima resistencia quiere decir que si se agrega el iesimo producto entonces se supera el valor de R o se aplasta algún producto de los que ya se eligieron.

Observar que el valor de máxima resistencia puede tomar un valor negativo únicamente si se agrega un producto con peso mayor a la máxima resistencia. Si el valor de máxima resistencia es negativo para alguna solución parcial, seguirá siendo negativo para cualquier solución sucesora. Esto sucede pues el valor de máxima resistencia se actualiza tomando el mínimo entre dos valores y las resistencias de los productos son entero no negativos.

Finalmente, el valor de máxima resistencia es utilizado para determinar si una solución candidata es factible o infactible. Dada una solución candidata, si su valor de máxima resistencia asociado es negativo entonces la solución candidata es inválida, en caso contrario es válida.

El objetivo de este trabajo es abordar el PJT 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 visitados 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,dado una secuencia ordenada S de n productos, el conjunto de soluciones está compuesto por todas las subsecuencias posibles de S.Para entender el funcionamiento se puede ver el Ejemplo 1.1 donde se analiza una instancia.

La idea del Algoritmo 1 para resolver el PJT es ir generando las soluciones de manera recursiva decidiendo en cada paso si un elemento de S es considerado o no y quedándose con la mejor solución de alguna de las dos ramas. Finalmente, al identificar una solución, determinar si es factible y óptima, y de ser así, devolver el tamaño del subconjunto solución.

En la Figura 1 se ve un ejemplo del árbol de recursión para la instancia vista en el Ejemplo 1.1. 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 candidatas (8 en este caso). La solución óptima $\{s2, s3\}$ está marcada en rojo y la otra solución factible $\{s1, s3\}$ en gris. Notar que la solución al problema original es exactamente $PJT\_FB(w, r, R, 0, 0, R, 0)$.

Algorithm 1 Algoritmo de Fuerza Bruta para PJT.

1: **function** $PJT\_FB(w,\ r,\ R,\ i,\ p,\ mr,\ k)$
2: **if** $i = n$ **then**
3: **if** $mr \geq 0 \wedge p \leq R$ **then return** k **else return** $-\infty$
4: $max\_resistencia \leftarrow \min\{mr - w_i, r_i\}$
5: **return** $\max\{PJT\_FB(w, r, R, i+1, p+w_i, max\_resistencia, k+1), PJT\_FB(w, r, R, i+1, p, mr, k)\}$.

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 contrario. Al haber generado todas las posibles soluciones, debe encontrarse la óptima.

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

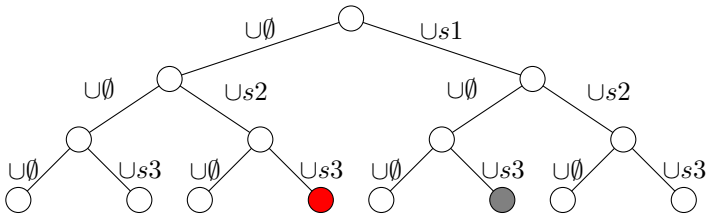


Figura 1: Ejecución del Algoritmo 1 para el Ejemplo 1.1.
En rojo la solución óptima $\{s2, s3\}$ y en gris la otra solución factible.

3. Backtracking

Los algoritmos de Backtracking siguen una idea similar a Fuerza Bruta pero con algunas consideraciones especiales. En esencia, se enumeran todas las soluciones formando un *árbol de backtracking* de manera similar a Fuerza Bruta, donde en cada nodo se generan todas las posibles decisiones locales y se mantiene la mejor solución hallada con alguna de ellas. La diferencia radica en las denominadas *podas* que son reglas que permiten evitar explorar partes del árbol en las que se *sabe* que no va a existir ninguna solución de interés. Generalmente estas podas dependen de cada problema en particular, pero las más comunes suelen dividirse en dos categorías: *factibilidad* y *optimalidad*.Para el PJT se utilizaran las 2 categorías mencionadas y se detallan a continuación.

Poda por factibilidad En este caso, una poda por factibilidad es la siguiente. Sea S' una solución parcial representada en un nodo intermedio n_0 tal que p es la suma de los pesos de los productos en S', y mr es la máxima resistencia que soportan los productos de S'. Como todos los números de S tienen pesos positivos, si $p > R$ entonces no va a haber ninguna forma de extender S' (o conservarlo) pues se supero el peso R que podía soportar el tubo. Además, si $mr < 0$ significa que se aplasto algún producto de S'(ver máxima resistencia 1.0.1), en este caso tampoco tiene sentido conservar o extender S'. De este modo, podemos evitar seguir explorando el subárbol formado debajo de n_0 y por lo tanto, reducir la cantidad de operaciones de nuestro algoritmo. Esta poda está expresada en la línea 9 del Algoritmo 2.

Poda por optimalidad Supongamos que ya se conoce una solución factible para el problema con cardinal K. Además, supongamos que se está en un nodo intermedio n_0 que representa a una solución parcial S' con k_1 elementos. Supongamos que k_2 es la cantidad de productos que todavía no fueron explorados, se calcula en la linea 10 del Algoritmo 2 . En este caso, si $k_1 + k_2 \leq K$, significa que a partir de S' no es posible mejorar la solución actual de cardinal K. Pues incluso si fuera posible agregar a S' todos los productos restantes, el cardinal del subconjunto resultante es a lo sumo K, es decir, cualquier solución factible generada a partir de S' va a ser a lo sumo tan buena como la que ya conocemos. Por lo tanto, se puede podar esta rama y así evitar el cómputo innecesario de operaciones. En la línea 5 del Algoritmo 2, se actualiza una variable global K cada vez que se encuentra una solución factible y se evalúa la regla de la poda en la línea 11.

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 las líneas 5, 9,10 y 11 solamente agrega un número constante de operaciones.Existe una familia de instancias para las cuales este algoritmo realiza la mayor cantidad de llamadas, es decir se reduce considerablemente la cantidad de podas, que son aquellas en las cuales hay varias combinaciones posibles de llegar al valor de R. Por otro lado, el mejor caso ocurre cuando la solución óptima se encuentra rápidamente. La familia de instancias con las siguientes características:

- La secuencia de pesos $w = \{R + 1, \ldots, R + 1, R\}$.
- La secuencia de resistencias $r = \{1, \ldots, 1\}$

Algorithm 2 Algoritmo de Backtracking para PJT.

```
1: K ← 0
2: function PJT_BT(w, r,p, i, mr, k)
3:     if i = n then
4:         if mr ≥ 0 ∧ p ≤ R then
5:             K ← máx{K, k}
6:             return k
7:         else
8:             return −∞
9:     if mr < 0 ∨ p > R then return −∞
10:    k_2 ← n − i
11:    if k + k_2 ≤ K then return −∞
12:    return máx{PJT_BT(w, r, p, i+1, mr, k), PJT_BT(w, r, p+w_i, mín{mr−w_i, r_i}, k+1)}.
```

- Con algún $R > 0$

logra una solución óptima al explorar la primera rama del árbol (ver Fig 1) y luego la poda por factibilidad garantiza que ningún otro nodo se va a ramificar. Por lo tanto, en estos casos el algoritmo se comporta de manera lineal y no se enumeran más de $O(n)$ nodos.

4. Programación Dinámica

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

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

Podemos definir $f(i,j)$: "máxima cantidad de productos que se pueden apilar de un subconjunto $\{S_i, \ldots, S_n\}$, cuando j es la máxima resistencia que se puede usar". Notar que $f(1,R)$ es la solución de nuestro problema. A continuación analizamos la correctitud de la función recursiva.

Correctitud

(i) Si $j < 0$ significa que en algún momento se agrego un producto que supero la máxima resistencia, de esta forma o bien se rompió el tubo, o bien se exploto algún producto. En cualquier caso, esta es una instancia invalida del problema, así que la respuesta es $f(i,j) = -\infty$.

(ii) Si $i = n+1$ entonces quiere decir que buscamos la máxima cantidad de productos del conjunto $\emptyset$, que podemos apilar cuando la máxima resistencia es $j \geq 0$. Por lo tanto, la respuesta es $f(i,j) = 0$.

(iii) En este caso, $i \leq n$ y $j \geq 0$ entonces estamos buscando máxima cantidad de productos que se pueden apilar de un subconjunto $S^i = \{S_i, \ldots, S_n\}$, cuando j es la máxima resistencia que se puede usar. Esta máxima cantidad, tiene que o bien apilar al i-ésimo producto o no apilarlo. Si no lo apila, entonces se tiene que buscar la máxima cantidad de productos que se puede apilar de S^{i+1} usando j como máxima resistencia, por lo tanto, debe encontrarse de manera recursiva $f(i+1,j)$. Si tiene al i-ésimo producto, entonces el resto de la solución debe usar como máxima resistencia exactamente $\text{mín}\{j - w_i, r_i\}$, utilizando los productos

de S^{i+1} y debe tener la mayor cantidad de productos entre todas ellas. Esto es precisamente $f(i+1, min\{j - w_i, r_i\})$ (ver maxima resistencia 1.0.1).
Por lo tanto, la mejor solución es $f(i,j) = \text{máx}\{f(i+1,j), 1+f(i+1, \text{mín}\{j-w_i, r_i\})$. Notar que al término de la derecha se le suma 1 por haber seleccionado al i-ésimo producto.

Memoización Observemos que la función recursiva (1) toma dos parámetros $i \in [1, \ldots, n]$ y $j \in [0, \ldots, R]$. Además los casos $i = n + 1$ o $j < 0$ son casos base y se resuelven en tiempo constante. Por lo tanto, la cantidad de *instancias* posibles con la que se puede llamar a la función, o combinación de parámetros, está determinada por la combinación de los mismos. En este caso, hay $\Theta(n * R)$ combinaciones posibles de parámetros.De esta manera podemos memoizar los resultados previamente calculados a fin de computar una sola vez cada instancia y asegurarnos de no resolver más de $\Theta(n * R)$ casos. El Algoritmo 3 muestra esta idea aplicada a la función (1). En la línea 5 se lleva a cabo el paso de memoización que solamente se ejecuta si la instancia no fue computada previamente.

Algorithm 3 Algoritmo de Programación Dinámica para PJT.

```
1: D_ij ← ⊥ for i ∈ [1, n], j ∈ [0, R].
2: function DP(i, j)
3:     if j < 0 then return −∞
4:     if i = n + 1 and j ≥ 0 then return 0
5:     if D_ij = ⊥ then D_ij ← máx{DP(i + 1, j), 1 + DP(i + 1, mín{j − w_i, r_i})}
6:     return D_ij
```

La complejidad del algoritmo entonces está determinada por la cantidad de instancias que se resuelven y el costo de resolver cada una de ellas. Como mencionamos previamente, a lo sumo se resuelven $O(n * R)$ instancias distintas, y como todas las líneas del Algoritmo 3 realizan operaciones constantes entonces cada instancia se resuelve en $O(1)$. Como resultado, el algoritmo tiene complejidad $O(n * R)$ en el peor caso. Es importante observar que el diccionario D 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 presenta los experimentos computacionales realizados para evaluar los distintos métodos presentados en las secciones anteriores. Las ejecuciones fueron realizadas en una notebook con CPU Intel Core i3 @ 2.3 GHz y 8 GB de memoria RAM, y utilizando el lenguaje de programación *C++*.

5.1. Métodos

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

- **FB**: Algoritmo 1 de Fuerza Bruta de la Sección 2.
- **BT**: Algoritmo 2 de Backtracking de la Sección 3.
- **BT-F**: Algoritmo 2 con excepción de la línea 11, es decir, solamente aplicando podas por factibilidad.
- **BT-O**: Similar al método BT-F pero solamente aplicando podas por optimalidad, o sea, descartando la línea 9 del Algoritmo 2.
- **DP**: Algoritmo 3 de Programación Dinámica de la Sección 4.

5.2. Instancias

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

- **densidad-alta**: Esta familia de instancias posee las siguientes características dado una secuencia de productos de tamaño n:
  - $R = n$
  - El peso de cada producto es el 20 % del valor R.
  - Inicializamos la secuencia de resistencias con el valor R en todas sus posiciones.

  . Con esto se necesitan pocos valores para alcanzar el valor de R.
- **densidad-baja**: Esta familia de instancias posee las siguientes características dado una secuencia de productos de tamaño n:
  - $R = n$
  - El peso de cada producto es el 5 % del valor R.
  - Inicializamos la secuencia de resistencias con el valor R en todas sus posiciones.

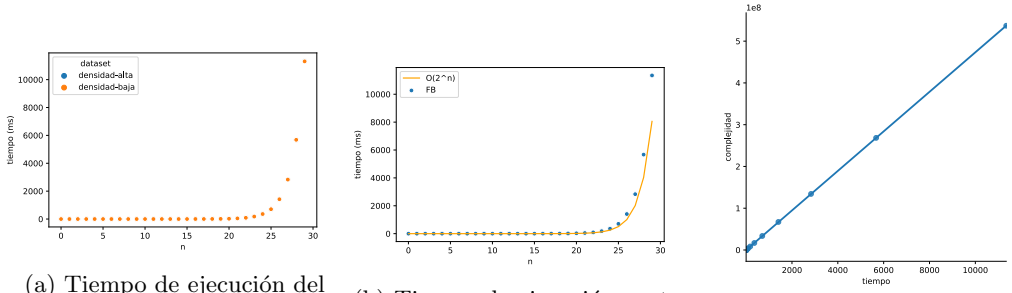
  Con esto se necesitan varios valores para alcanzar el valor de R.
- **bt-mejor-caso**: Son las instancias para el mejor caso de Backtracking definidas en la Sección 3. Esta familia de instancias posee las siguientes características dado una secuencia de productos de tamaño n:
  - La secuencia de pesos $w = \{R + 1, \ldots, R + 1, R\}$.
  - La secuencia de resistencias $r = \{1, \ldots, 1\}$
  - Con algún $R > 0$
- **Caso-medio**: Esta familia de instancias posee las siguientes características dado una secuencia de productos de tamaño n:
  - La secuencia de pesos $w = \{1, \ldots, 1\}$.
  - La secuencia de resistencias $r = \{R, \ldots, R\}$
  - Con $R = 2 * n$
- **bt-peor-caso**: Es el dataset correspondiente a densidad-baja.
- **dinamica**: Esta familia de instancias tiene instancias con distintas combinaciones de valores para n y R en los intervalos $[1000, 8000]$. Los valores de w y r son una permutación de el conjunto $\{1, \ldots, n\}$.

5.3. Experimento 1: Complejidad de Fuerza Bruta

En este experimento se analiza la performance del método FB en distintas instancias. Vamos contrastar empíricamente las afirmaciones mencionadas en la seccion 2, evaluaremos el metodo FB utilizando los datasets densidad-alta y densidad-baja.

En la Figura 2a se ilustra los resultados del experimento, donde podemos apreciar que ambas curvas están solapadas para la todas las instancias, tanto que la diferencia es casi imperceptible, si observamos fijamente para el valor de n=30 se observa una pequeña diferencia entre ambos datasets.Con lo expuesto en los gráficos podemos concluir que los tiempos de ejecución parecen no alterarse para cada dataset con sus respectivas características y seguir la misma curva de crecimiento.

Mientas que en la Figura 2b se ilustra el tiempo de ejecución de FB a la par de una función exponencial de $O(2^n)$, los cuales se adecuan a la misma. Finalmente, para la Figura 2c se enumeran las instancias para cada una se gráfica el tiempo de ejecución real contra el tiempo esperado, es decir, su *gráfico de correlación*.

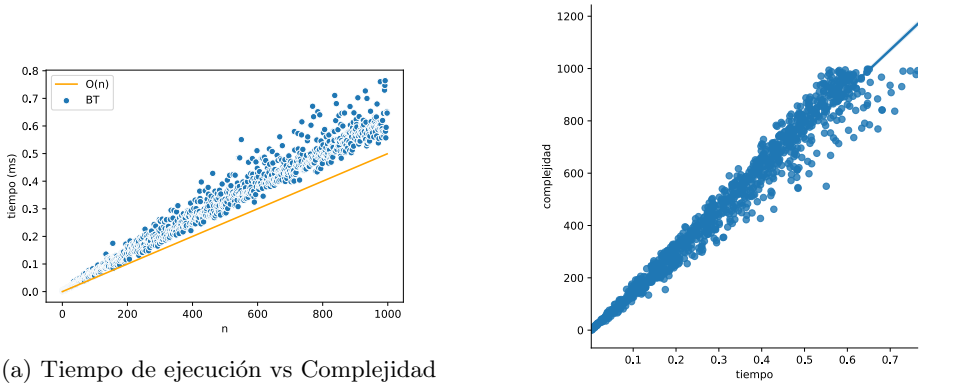


(a) Tiempo de ejecución del método FB sobre densidad-alta y densidad-baja.

(b) Tiempo de ejecución contra complejidad esperada.

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

Figura 2: Análisis de complejidad del método FB.



(a) Tiempo de ejecución vs Complejidad esperada.

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

Figura 3: Análisis de complejidad del método BT para el data set bt-mejor-caso.

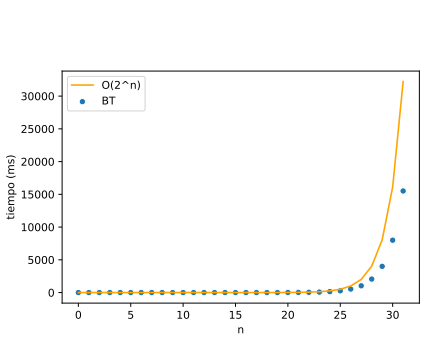
Se puede ver que el tiempo de ejecución sigue claramente una curva exponencial y además la correlación con la función 2^n es positiva. En particular, el índice de correlación de Pearson de ambas variables es $r \approx 0{,}999999$. Por lo tanto, podemos afirmar que el algoritmo se comporta como se describió inicialmente en las hipótesis.

5.4. Experimento 2: Complejidad de Backtracking

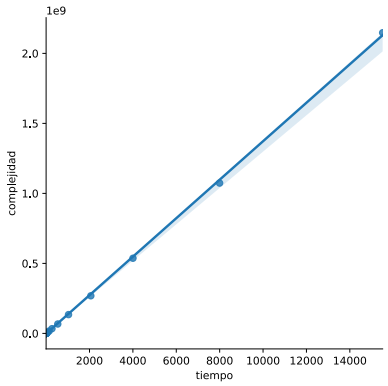
En esta experimentación vamos a contrastar las hipótesis de la Sección 3 con respecto a las familias de instancias de mejor y peor caso para el Algoritmo 2, y su respectiva complejidad. Para esto evaluamos el método BT con respecto los datasets bt-mejor-caso y bt-peor-caso.

En la Figura3a y muestran los gráficos de tiempo de ejecución de BT y de correlación para cada dataset respectivamente. Para las instancias de mejor caso se puede notar que la serie de puntos muestra un crecimiento lineal. Ademas en la figura3b se observa una correlación positiva entre los tiempos de ejecución y una función lineal. El índice de correlación de Pearson es $r \approx 0{,}981234$ lo cual confirma la complejidad estimada.

Por otra parte, en la figura4 para las instancias de peor caso los tiempos de ejecución se presentan más ajustados a la curva de complejidad exponencial. A pesar de que no se puede asegurar que se recorra el árbol de bactraking completamente, en la practica para estas instancias el comportamiento del algoritmo efectivamente demuestra un comportamiento exponencial. Notemos que en este caso se ejecutaron instancias hasta $n = 30$, número elegido para evitar que el tiempo de ejecución sea demasiado grande (> 10 segundos). Para estas instancias el índice de correlación



(a) Tiempo de ejecución vs Complejidad esperada.



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

Figura 4: Análisis de complejidad del método BT para el data set bt-peor-caso.

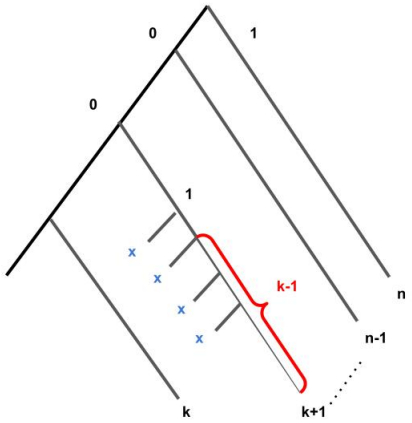


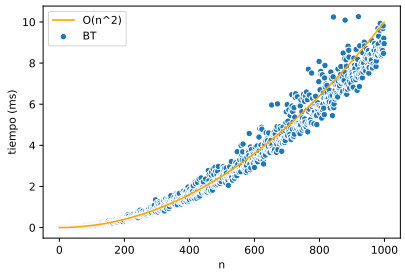
Figura 5: Esquema de poda optimalidad en el arbol de backtracking.

de Pearson es de $r \approx 0{,}999884$ contra una función exponencial con base 2.

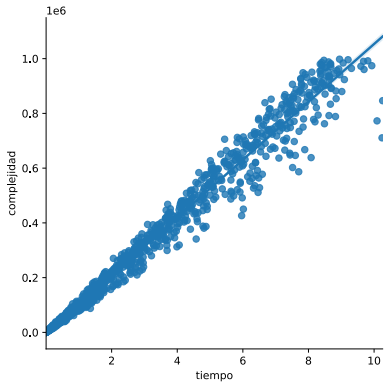
5.5. Experimento 3: Caso Medio Bactraking

Para este experimento se considera por hipótesis que al tener la posibilidad de agregar todos los productos en el tubo, el algoritmo 2 recorrerá todos los nodos del árbol de backtracking, logrando un comportamiento exponencial. Se evalúa el método BT con el dataset bt-caso-medio obteniendo los resultados que se ilustran en la Figura 6a.

Los resultados obtenidos son distintos a los esperados, en particular, en la Figura 6a se observa un comportamiento cuadrático para este tipo de entradas. Como se puede analizar en la figura 5, lo que sucede es que al recorrer por completo cada rama derecha, se obtiene una solución con un elemento más que la que se tenía hasta entonces. Luego si se considera la rama que puede agregar k+1 elementos si se recorre hasta las hojas, se observa que cualquier decisión de no agregar un elemento activará la poda por optimalidad. Esto sucede pues con los elementos restantes, k-1 en este caso, no es posible mejorar la solución de valor k que se encontró hasta el momento.Salvo en la rama del árbol que no agrega ningún elemento, esta situación se repite ante cualquier decisión de no agregar un elemento, logrando de esta forma una complejidad cuadrática para el algoritmo sobre esta familia de instancias. En la Figura 6b se puede observar la correlación entre el tiempo

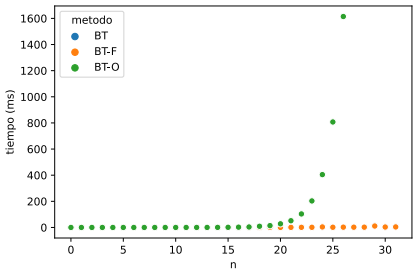


(a) Tiempo de ejecución vs Complejidad esperada.

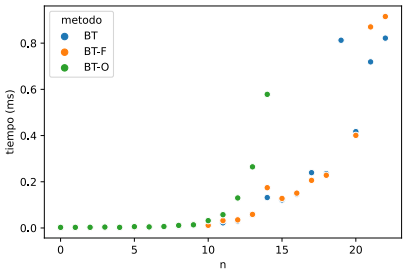


(b) Gráfico de correlación entre el tiempo de ejecución y la complejidad esperada.

Figura 6: Análisis de complejidad del método BT para el data set bt-caso-medio.



(a) Efectividad de las podas para densidad-alta.



(b) Efectividad de las podas con zoom para densidad-alta.

Figura 7: Comparación de efectividad en las podas densidad-alta

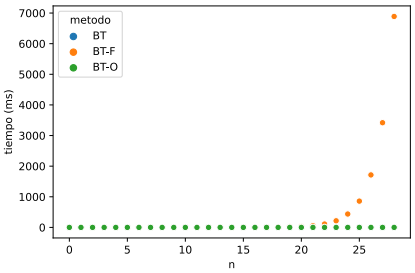
de ejecución y una funcion cuadratica, la cual es bastante positiva y es confirmado por el índice de correlación de Pearson que es $r \approx 0{,}988099$.

5.6. Experimento 4: Efectividad de las podas

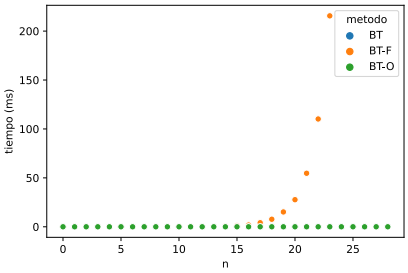
En este experimento se compara el funcionamiento de los métodos BT, BT-F y BT-O con respecto a los datasets densidad-alta, densidad-baja y caso-medio. Queremos observar cual es el comportamiento de las podas para distintos datasets con determinadas características, al alcanzar su correspondiente solución.

5.6.1. Dataset Densidad-Alta

En la Figura 7 se ilustran los resultados para este tipo de entradas. Analizando el gráfico 7b a partir de las entradas de tamaño n=20, se puede observar que la poda por optimalidad tuvo muy poco impacto en el comportamiento del algoritmo. Como la solución al problema utiliza muy pocos productos, digamos k, el algoritmo explorara todos los subconjuntos con cardinal mayor a k. y la poda por optimalidad tardara mucho en activarse. En cambio la poda por factibilidad se activa inmediatamente al generar subconjuntos de tamaño mayor a k, y como k es un valor chico en estas instancias, esta poda resulta mas efectiva.

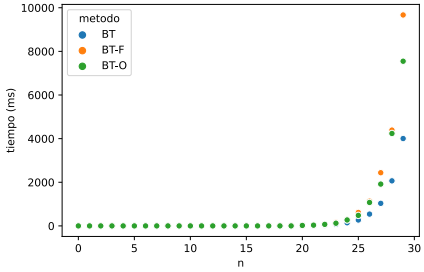


(a) Efectividad de las podas caso medio.

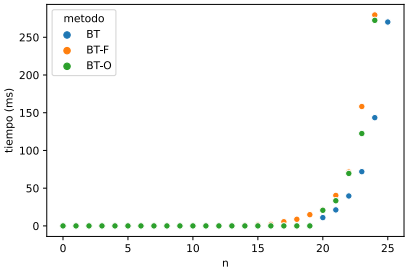


(b) Efectividad de las podas con zoom caso medio.

Figura 8: Comparación de efectividad en las podas Dataset caso-medio.



(a) Efectividad de las podas para densidad-baja.



(b) Efectividad de las podas con zoom para densidad-baja.

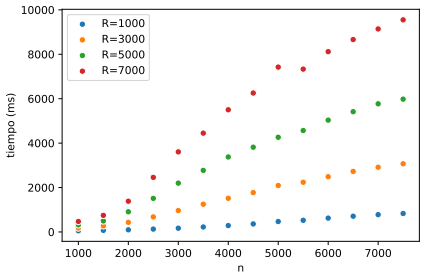
Figura 9: Comparación de efectividad en las podas densidad-baja.

5.6.2. Dataset Caso-Medio

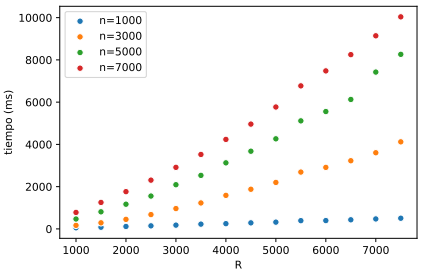
Finalmente en la Figura 8 se tiene los resultados para este dataset,en el cual podemos notar que la poda optimalidad es la que tuvo menos impacto en el algoritmo. Esto se debe a que siempre exploramos soluciones parciales que dejan de ser factibles en cierto punto, por ejemplo dado cierta rama del arbol de bactraking, si la solución parcial S' posee cardinal k en el nodo i, la profundidad del árbol para esta rama es $n - i$ y el máximo global es mayor a $k + n - i$ con lo cual no va a ser una solución factible pero al no estar activa la poda de optimalidad, se termina recorriendo todas las ramas que posean estas características, resultando en una perfomance muy similar a la de Fuerza Bruta.

5.6.3. Dataset Densidad-Baja

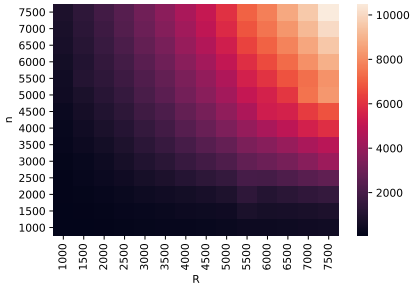
En la Figura 9 se muestran los resultados obtenidos para este dataset, puede observar que para el dataset densidad-baja la diferencia entre activar o no las podas generó muy poco efecto y se mantiene la misma linea de comportamiento para los distintos métodos. Esto es debido a que el dataset densidad baja se compone de entradas que conforman el peor caso analizado en Experimento 2, Complejidad de Bactraking. Para este tipo de instancias el algoritmo de BT se comporta de manera similar a FB, con lo cual podemos afirmar el algoritmo de BT no una buena eleccion para para este tipo de instancias.



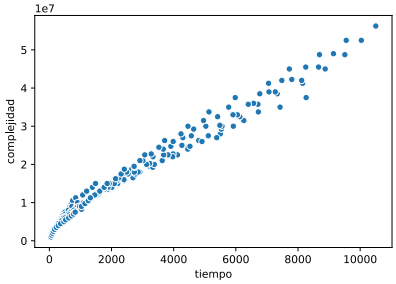
(a) Tiempo de ejecución en función de n.



(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 10: Resultados computacionales para el método DP sobre el dataset dinamica.

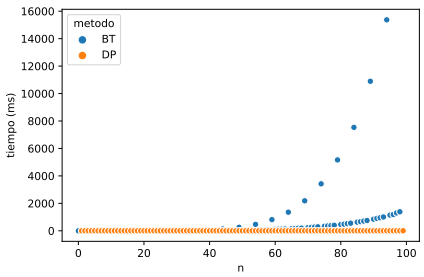
5.7. Experimento 5: Complejidad de Programación Dinámica

A continuación se analiza la complejidad del algoritmo de Programación Dinámica en la práctica y su correlación con su cota teórica calculada en la Sección 4. Para este experimentos se utilizaran las instancias del dataset dinámica sobre el método DP.

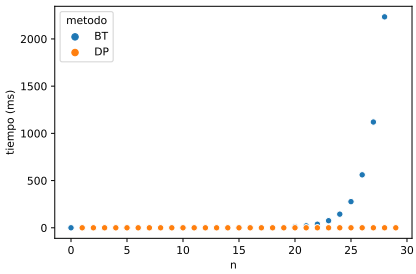
Las Figuras 10a y 10b muestran el crecimiento del tiempo de ejecución en función de n y R respectivamente. En la figura10a se fija la variable R y podemos notar que los tiempos de ejecución en función de n, crecen de manera lineal. De igual manera, en la figura10b podemos observar que fijando la variable n, se obtiene un comportamiento similar, es decir, los tiempos de ejecución en función de R poseen un crecimiento lineal . Esto se reafirma en la Figura 10c donde se muestra el crecimiento del tiempo de ejecución en función de ambas variables al mismo tiempo. Podemos apreciar que el crecimiento es similar tanto en la dirección de n como en la de R. Finalmente, para confirmar que el tiempo de ejecución de nuestro algoritmo es efectivamente $O(nR)$, se exhibe un gráfico de correlación a lo largo de todas las instancias comparando el tiempo de ejecución contra el tiempo esperado. Este gráfico muestra una correlación muy positiva entre ambas series de datos, lo cuál es confirmado por el índice de correlación de Pearson que es $r \approx 0{,}990276$.

5.8. Experimento 6: Backtracking vs Programación Dinámica

Para finalizar, presentamos un experimento que compara dos técnicas algorítmicas distintas, las cuales son Bactraking y DP. La idea es comprobar la eficacia de ambas para el PJT,entender el comportamiento de cada una sometiendo a pruebas para distintos datasets y en base a ello poder tomar la mejor decisión al momento de elegir alguna. Nuestra hipótesis es que ambos algoritmos van a comportarse mejor en situaciones diferentes. Por ejemplo, Backtracking funciona muy bien en las instancias del tipo caso-medio, y sus podas pueden lograr una buena performance, por otro lado ante el crecimiento de la variable R es posible que la programación dinámica no sea una opción viable al tener que mantener su estructura de memoizacion.

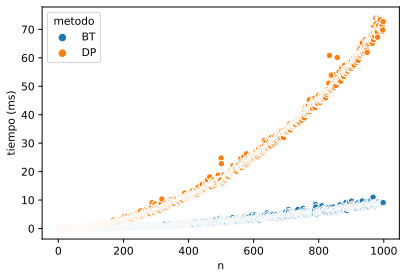


(a) Dataset densidad-alta.

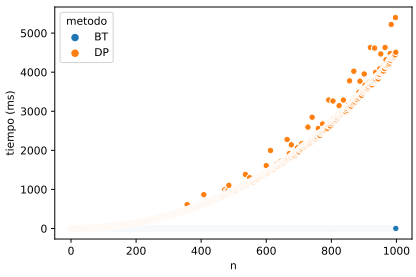


(b) Dataset densidad-baja.

Figura 11: Comparación de tiempos de ejecución entre DP y BT.



(a) Dataset caso-medio.



(b) Dataset mejor-caso.

Figura 12: Comparación de tiempos de ejecución entre DP y BT.

Una observación importante es que ningún algoritmo *domina* al otro en términos de complejidad. Dicho de otro modo, no es cierto que $O(2^n) \subseteq O(nR)$ ni tampoco que $O(nR) \subseteq O(2^n)$.

En la Figura 11 se muestra los tiempos de ejecucion de los metodos BT y DP sobre los datasets densidad-alta y densidad-baja, podemos ver que para los datasets de densidad alta y baja la programación dinámica es muy efectiva, considerando que el valor de R esta controlado.Se puede observar que el comportamiento de BT en las instancias de densidad baja es rápidamente exponencial con lo cual PD puede ser una opción segura. Por otro lado en la figura 12 se ilustra como el comportamiento de programación Dinámica puede cambiar rápidamente. En este experimento se usó los dataset caso-medio y mejor-caso en el que BT se comporta muy bien, en cambio DP se ve afectado por el valor de R=2n, pues solo el costo de creación de la estructura de memoización tiene una complejidad estimada similar al de la ejecución del algoritmo de BT para esta instancia. Con lo cual es evidente que mantener una estructura de memoización, para evitar repetir cálculos, no siempre se traduce en un algoritmo mejor performance.Finalmente pudimos verificar que nuestras hipótesis son ciertas, para los datasets densidad alta y baja, PD posee una mejor performance mientras que para los datasets caso-medio y mejor-caso, pierde su efectividad ante el crecimiento de la variable R.

6. Conclusiones

En este trabajo presentamos tres algoritmos que usan técnicas distintas para resolver el PJT. El algoritmo de fuerza bruta demostró ser poco eficiente sin importar el tipo de instancia a resolver, con lo cual resulta una opción poco viable en la practica. Por otro lado el algoritmo de Bactraking introduce podas, que como pudimos contrastar, permiten mejorar notablemente los tiempos de ejecución para determinadas instancias. Sin embargo aunque son efectivas, las podas no siempre

garantizan una mejora respecto del algoritmo de fuerza bruta, esto se vio por ejemplo en el dataset densidad-baja. Por ultimo, el algoritmo de programación dinámica demuestra una buena performance ante el crecimiento de n mostrando una mejora considerable respecto a las técnicas anteriores frente a determinados datasets. Hay que tener en cuenta que ante la combinación de valores grandes de n y R, el costo de memoizar puede ser excesivo. Esto ultimo podemos tomarlo como problema futuro a analizar, sobre los costos de memoria para dicha estructura y sus efectos en la practica.