DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Ejemplo de TP 1: Subset Sum

Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Mozart, Wolfgang Amadeus	K310/I	wamozart@dc.uba.ar

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

Observaciones

- Esta es una resolución de ejemplo de un trabajo práctico de cuatrimestres anteriores. El objetivo es presentar un *posible* modelo de informe, descripción de algoritmos, presentación de experimentos y análisis. Sin embargo, existen varias maneras de lograr el mismo resultado que también son correctas.
- La resolución puede contener algunos errores. De todos modos, recordar que el foco está puesto en la metodología de resolución del problema y de comunicación. Se recomienda leer cada sección y luego ver el código de LaTeX asociado para repasar o aprender cómo escribir fórmulas matemáticas, generar figuras, hacer referencias a secciones y organizar el documento en general.
- Recordar que la idea es que cada uno escriba el informe de manera personal, ya que es la única manera de aprender a comunicar ideas. Se recomienda leer este ejemplo, cerrarlo y luego comenzar a escribir el trabajo.

1. Introducción

El *problema de suma de subconjuntos* (SSP, por sus siglas en inglés) es uno de los problemas fundamentales de las Ciencias de la Computación. Formalmente, dado un conjunto de $n \geq 0$ números enteros positivos $S = \{s_1, s_2, \ldots, s_n\}$ y otro entero positivo $W \in \mathbb{N}$, el SSP consiste en decidir si existe un subconjunto $S' \subseteq S$ que sume exactamente W, es decir, $\sum_{x \in S'} x = W$. Para simplificar las explicaciones, consideramos *solución* a todo subconjunto $S' \subseteq S$ y decimos que es *factible* si su suma es W. En este trabajo se va a resolver la variante de optimización, que además indica cuál es el **mínimo** cardinal de un conjunto solución factible S'. Para simplificar, decimos que si no existe ningún conjunto factible, la respuesta es ∞.

A continuación se exhiben algunos ejemplos con sus correspondientes respuestas esperadas. Si $S = \{1, 2, 3, 4, 5\}$ y $W = 5$ entonces existen 3 soluciones factibles $S_1 = \{1, 4\}$, $S_2 = \{2, 3\}$ y $S_3 = \{5\}$ y la de menor cardinalidad es S_3, por lo tanto la respuesta es 1. Por otra parte, si $S = \{2, 4, 6, 8, 10\}$ y $W = 7$ entonces no existe ninguna solución factible, dado que todos los números de S son pares y W es impar. Por lo tanto, la respuesta en este caso es ∞.

El objetivo de este trabajo es abordar el SSP 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).

El trabajo va a estar ordenado de la siguiente manera: primero en la Sección 2 se define el algoritmo recursivo de Fuerza Bruta para recorrer todo el conjunto de soluciones y se analiza su complejidad. Más tarde, en la Sección 3 se explica el algoritmo de Backtracking con un breve análisis de mejores y peores casos. Luego, se introduce el algoritmo de DP en la Sección 4 junto con la demostración correspondiente de correctitud y un análisis de complejidad. Finalmente, en la Sección 5 se presentan los experimentos computacionales con sus respectiva discusión, y las conclusiones finales se encuentran en la Sección 6.

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 subconjuntos de S, es decir, es el *conjunto de partes de* S que se escribe $\mathcal{P}(S)$.

Por ejemplo, si $S = \{1, 2, 3\}$ y $W = 3$, el conjunto $\mathcal{P}(S) = \{\ \emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\ \}$, y el conjunto de soluciones factibles es $\{\{1, 2\}, \{3\}\}$.

La idea del Algoritmo 1 para resolver el SSP 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 de ser así, devolver el cardinal de esa solución.

En la Figura 1 se ve un ejemplo del árbol de recursión para la instancia $S = \{1, 2, 3\}$ y $W = 3$. Cada nodo intermedio del árbol representa una *solución parcial*, es decir, cuando aún no se tomaron todas las decisiones de qué elementos incluir, mientras que las hojas representan a todas las soluciones (8 en este caso). La solución óptima $\{3\}$ está marcada en rojo y la otra solución factible $\{1, 2\}$ en gris. Notar que la solución al problema original es exactamente $FB(S, W, 0, 0, 0)$.

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

Algorithm 1 Algoritmo de Fuerza Bruta para SSP.

```
1: function FB(S, W, i, w, k)
2:     if i = n then
3:         if w = W then return k else return ∞
4:     return mín{FB(S, W, i + 1, w, k), FB(S, W, i + 1, w + S_i, k + 1)}.
```

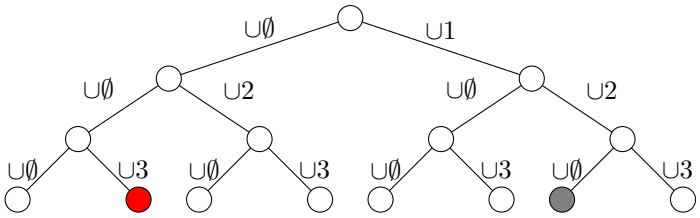


Figura 1: Ejemplo de ejecución del Algoritmo 1 para $S = \{1, 2, 3\}$ y $W = 3$.
En rojo la solución óptima $\{3\}$ y en gris la otra solución factible.

en el caso contrario. Al haber generado todas las posibles soluciones, debe encontrarse la óptima (de existir).

La complejidad del Algoritmo 1 para el peor caso es $O(2^n)$. Esto es así, porque el árbol de recursión es un árbol binario completo de $n+1$ niveles (contando la raíz), dado que cada nodo se ramifica en dos hijos y en cada paso el parámetro 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 suma, mínimos y comparaciones. Como corolario, se puede concluir que el algoritmo se comporta de igual manera frente a todos los tipos de instancia, dado que siempre 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.

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*.

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 con suma $w = \sum S'$. Claramente, al ser todos los números de S enteros positivos, si $w > W$ entonces no va a haber ninguna forma de extender S' (o conservarlo) de manera tal que su suma sea W. 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 6 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 elementos. En este caso, si $k \geq K$, como cualquier decisión que se tome a continuación en el subárbol va a agregar o mantener la cantidad de elementos seleccionados, podemos asegurar que cualquier solución factible que se encuentre al explorarlo va a ser al menos tan buena como la que ya conocemos. 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 K cada vez que se halla una solución factible en la línea 4, y se evalúa la regla de la poda en la línea 7.

Algorithm 2 Algoritmo de Backtracking para SSP.

```
1: K ← ∞
2: function BT(S, W, i, w, k)
3:     if i = n then
4:         if w = W then K ← mín{K, k}
5:         if w = W then return k else return ∞
6:     if w > W then return ∞
7:     if k ≥ K then return ∞
8:     return mín{BT(S, W, i + 1, w, k), BT(S, W, i + 1, w + S_i, k + 1)}.
```

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 4, 6 y 7 solamente agrega un número constante de operaciones. Existe una familia de instancias para las cuales este algoritmo va a enumerar todo el árbol, que son aquellas con $S = \{1, \ldots, 1\}$ ($|S| = n$) y $W = n$. Notar que existe una única solución factible que es tomar todo S y esto solamente ocurre en el último nodo explorado (ver Fig 1). Por otro lado, el mejor caso ocurre cuando la solución óptima se encuentra rápidamente. La familia de instancias $S = \{1, \ldots, 1, W\}$ con algún $W > 0$ una solución óptima al explorar la primera rama (ver Fig 1) y luego la poda por optimalidad garantiza que ningún otro nodo se va a ramificar. Por lo tanto, en estos casos el algoritmo se comporta de manera lineal. Observar que existen otras familias de instancias para las cuales el algoritmo también tiene un comporamiento similar, por ejemplo, si $S = \{W + 1, \ldots, W + 1\}$ con algún $W > 0$, entonces la poda por factibilidad garantiza que no se enumeran más de $O(n)$ nodos.

4. Programación Dinámica

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

$$f(i, w) = \begin{cases} \infty & \text{si } w > W, \\ \infty & \text{si } w \neq W \wedge i = n + 1, \\ 0 & \text{si } w = W \wedge i = n + 1, \\ \text{mín}\{f(i + 1, w), 1 + f(i + 1, w + S_i)\} & \text{caso contrario.} \end{cases} \tag{1}$$

Coloquialmente, podemos definir $f(i, w)$: "mínimo cardinal de un subconjunto de $\{S_i, \ldots, S_n\}$ que sume $W - w$". Claramente, $f(1, 0)$ es "mínimo cardinal de un subconjunto de S que sume W" lo cuál es exactamente la solución de nuestro problema. Veamos que la recursión es efectivamente lo que dice su definición coloquial.

Correctitud

(i) Si $w > W$ entonces claramente ningún subconjunto va a sumar a $W - w < 0$ ya que todos los números son enteros positivos, así que la respuesta es $f(i, w) = \infty$.

(ii) Si $i = n + 1$ entonces quiere decir que buscamos subconjuntos de $\emptyset$ que sumen $W - w$. Si $w \neq W$ entonces buscamos subconjuntos de $\emptyset$ que sumen algo distinto de 0, lo cual es imposible. Por lo tanto, la respuesta es $f(i, w) = \infty$.

(iii) Análogamente, si $i = n+1$ entonces quiere decir que buscamos subconjuntos de $\emptyset$ que sumen $W - w$. En este caso, $w = W$ entonces buscamos subconjuntos de $\emptyset$ que sumen 0, y por lo tanto como $\emptyset \subseteq \emptyset$ la respuesta es $f(i, w) = 0$, ya que es el cardinal de la solución.

(iv) En este caso, $i \leq n$ y $w < W$ entonces estamos efectivamente buscando un subconjunto de $S^i = \{S_i, \ldots, S_n\}$ que sume $W' = W - w > 0$. De existir un subconjunto, tiene que o bien tener al i-ésimo elemento o no tenerlo. Si no lo tiene, entonces tiene que ser a su vez un subconjunto de S^{i+1} y sumar W', por lo tanto, debe encontrarse de manera recursiva $f(i+1, w)$. Si tiene al i-ésimo elemento, entonces el resto de la solución debe sumar exactamente $W' - S_i$, utilizando elementos de S^{i+1} y debe ser la de mínimo cardinal entre todas ellas. Esto es precisamente $f(i+1, w+S_i)$. Por lo tanto, la mejor solución es $f(i, w) = \text{mín}\{f(i+1, w), 1 + f(i+1, w+S_i)\}$. Notar que al término de la derecha se le suma 1 por haber seleccionado al i-ésimo elemento.

Memoización Notemos que la función recursiva (1) toma dos parámetros $i \in [1, \ldots, n]$ y $w \in [0, \ldots, W]$. Notar que los casos $i = n+1$ o $w > W$ 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 $\Theta(n * W)$ combinaciones posibles de parámetros. 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 ellos y asegurarnos no resolver más de $\Theta(n * W)$ casos. El Algoritmo 3 muestra esta idea aplicada a la función (1). En la línea 6 se lleva a cabo el paso de memoización que solamente se ejecuta si el estado no había sido previamente computado.

Algorithm 3 Algoritmo de Programación Dinámica para SSP.

```
1: M_iw ← ⊥ for i ∈ [1, n], w ∈ [0, W].
2: function DP(i, w)
3:     if w > W then return ∞
4:     if i = n + 1 and w ≠ W then return ∞
5:     if i = n + 1 and w = W then return 0
6:     if M_iw = ⊥ then M_iw ← mín{DP(i + 1, w), 1 + DP(i + 1, w + S_i)}
7:     return M_iw
```

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 $O(n * w)$ estados distintos, y como todas las líneas del Algoritmo 3 realizan operaciones constantes entonces cada estado se resuelve en $O(1)$. Como resultado, el algoritmo tiene complejidad $O(n * w)$ en el peor caso. Es importante observar que el diccionario M se puede implementar como una matriz con acceso y escritura constante. Más aún, notar que su inicialización tiene costo $\Theta(n * w)$, por lo tanto, el mejor y peor caso de nuestro algoritmo va a tener costo $\Theta(n * w)$.

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 workstation con CPU Intel Core i7 @ 2.8 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 7, es decir, solamente aplicando podas por factibilidad.
- **BT-O**: Similar al método BT-F pero solamente aplicando podas por optimalidad, o sea, descartando la línea 6 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. Primero, antes de enumerar los *datasets*, se define la *densidad* de una instancia como el cociente $\frac{\text{máx}\, S_i}{W}$, es decir, es una medida de cuántos números de S se necesitan para sumar W. A menor densidad, los números de S son más chicos en relación a W y por lo tanto se necesitan más de ellos. Finalmente, los *datasets* definidos se enumeran a continuación.

- **densidad-alta**: En esta familia cada instancia tiene los números $1, \ldots, n$ en S en algún orden aleatorio y se toma $W = \frac{n}{2}$.
- **densidad-baja**: Para esta conjunto de instancias se toman los números $1, \ldots, n$ en S en algún orden aleatorio y se toma $W = \frac{n(n-1)}{4}$, es decir, la mitad de la suma de todos los números.
- **bt-mejor-caso**: Cada instancia de n elementos, está formada por $S = \{1, \ldots, 1, W\}$ y algún $W > n$. Son las instancias para el mejor caso de Backtracking definidas en la Sección 3.
- **bt-peor-caso**: Cada instancia de n elementos, está formada por $S = \{1, \ldots, 1, 1\}$ y $W = n$. Son las instancias para el peor caso de Backtracking definidas en la Sección 3.
- **dinamica**: Esta familia de instancias tiene instancias con distintas combinaciones de valores para n y W en los intervalos $[1000, 8000]$. Los números en S son una permutación de el conjunto $\{1, \ldots, n\}$.

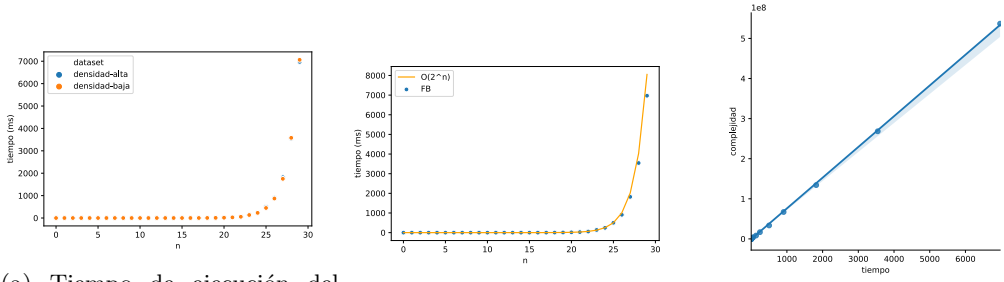
5.3. Experimento 1: Complejidad de Fuerza Bruta

En este experimento se analiza la performance del método FB en distintos contextos. El análisis de complejidad 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 y densidad-baja y se grafica los tiempos de ejecución en función de n.

La Figura 2a presenta los resultados del experimento, donde se puede apreciar que ambas curvas están solapadas para la mayoría de 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, $O(2^n)$. En la Figura 2b se ilustra el tiempo de ejecución de FB a la par de una función exponencial de $O(2^n)$. Por otro lado, para la Figura 2c se enumeran las instancias $I_1, \ldots, I_k$ y para cada una se grafica el tiempo de ejecución real $T(I_i)$ contra el tiempo esperado $E(I_i) = 2_i$, es decir, su *gráfico de correlación*.

Se puede ver que el tiempo de ejecución sigue claramente una curval 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 $r \approx 0{,}999893$. Por lo tanto, podemos afirmar que el algoritmo se comporta como se describió inicialmente en las hipótesis.

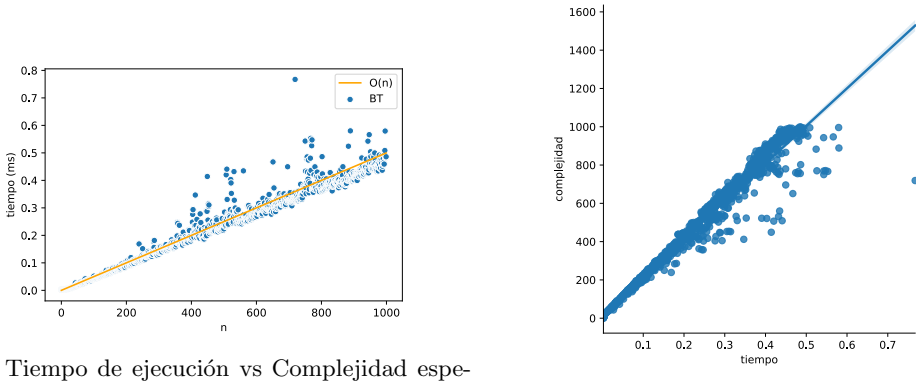


(a) Tiempo de ejecución del método FB sobre densidad-alta y densidad-baja.

(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.

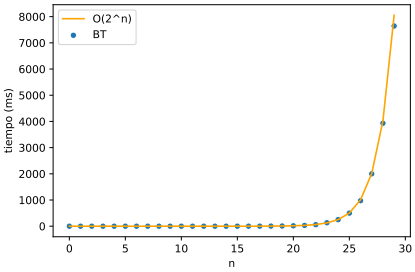
Figura 3: Análisis de complejidad del método BT para el data set bt-mejor-caso.

5.4. Experimento 2: Complejidad de Backtracking

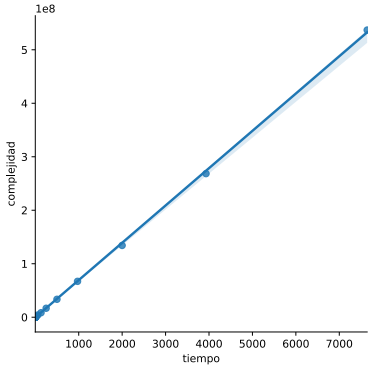
En esta experimentación vamos a contrastar las hipótesis de la Sección 3 con respecto a las familias de instancias de mejor y peor caso para el Algoritmo 2, y su respectiva complejidad. Para esto evaluamos el método BT con respecto los datasets bt-mejor-caso y bt-peor-caso.

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 $n = 1000$. 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 $r \approx 0{,}973844$ 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 peor caso no se ve este comportamiento, y los tiempos de ejecución se presentan más ajustados a la curva de complejidad 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 de Pearson es de $r \approx 0{,}999891$ contra una función exponencial con base 2.



(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.

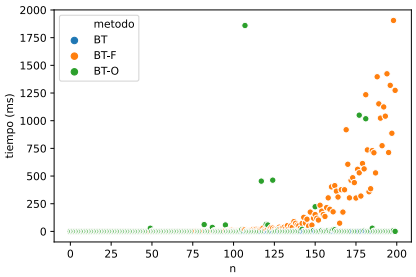
5.5. Experimento 3: Efectividad de las podas

Naturalmente, surgen varias preguntas luego del experimento anterior. En particular, una de ellas es qué sucede en el medio del comportamiento lineal y exponencial, y qué factores afectan al algoritmo para ir transitando entre ambos escenarios. Una de las hipótesis es que el Algoritmo 2 mejora su funcionamiento dependiendo de la densidad de las instancias, es decir, de cuántos elementos de S se necesitan para sumar W. Por ejemplo, si $W = 100$ y $S_i > 50$ para todo i, entonces al seleccionar dos elementos en una solución parcial del algoritmo, o bien se suma W o bien se supera ese valor y la poda por factibilidad es ejecutada. Por otra parte, esto también incide en el funcionamiento de la poda por optimalidad ya que al encontrarse una solución con cardinal C, la altura del árbol de backtracking se reduce a C.

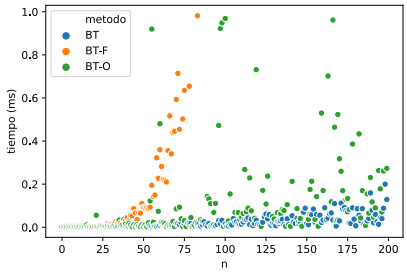
En este experimento se compara el funcionamiento de los métodos BT, BT-F y BT-O con respecto a los datasets densidad-alta y densidad-baja. La hipótesis es que para las instancias de alta densidad los algoritmos van a ser más eficientes que con aquellas de baja densidad.

En la Figura 5 se muestra los resultados para el dataset densidad-alta. Se ejecutaron instancias hasta $n = 200$ para evitar que el algoritmo demore más de unos segundos. Una observación interesante es que los tiempos de ejecución para los tres métodos están entre aquellos observados para los mejores y peores casos del algoritmo de BT. Por otra parte, es interesante que la poda por optimalidad tuvo mayor impacto que la poda por factibilidad. Esto puede deberse al hecho de que la densidad fue definida con respecto al máximo elemento S pero sin tener en cuenta el resto, por lo tanto, puede ser que existan varios elementos que juntos sean menores a W afectando la efectividad de la poda por factibilidad. Es interesante, sin embargo, mirar la Figura 5b que hace un acercamiento para evaluar mejor la diferencia entre BT y BT-O. En ella se aprecia que la combinación de ambas podas es más efectiva y es por esto que se puede concluir que la poda por factibilidad tiene un impacto positivo en el algoritmo final.

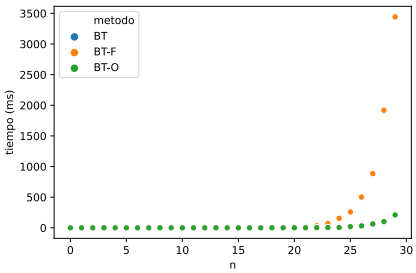
Por el lado de las instancias de baja densidad, los resultados están expuestos en la Figura 5c. El mensaje principal de estos gráficos es que a diferencia de las instancias de densidad alta, la efectividad de las podas no es lo suficientemente importante como para modificar de manera considerable el comportamiento del algortimo. En particular, esto se observa en el tamaño de las instancias ejecutadas (hasta $n = 30$) que si bien presentan tiempos más chicos que en las instancias de peor caso conservan su naturaleza exponencial. Como conclusión podemos afirmar que la densidad de las instancias tiene un peso significativo en el funcionamiento de este algoritmo.



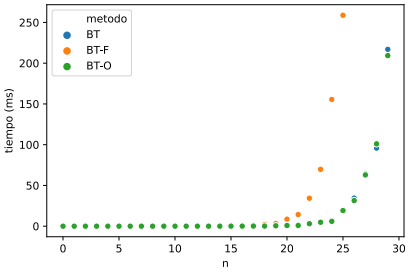
(a) Efectividad de las podas para densidad-alta.



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



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



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

Figura 5: Comparación de efectividad en las podas.

5.6. Experimento 4: Complejidad de Programación Dinámica

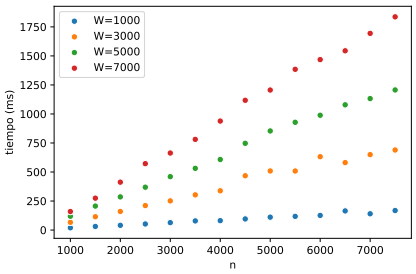
A continuación se analiza la eficiencia del algoritmo de Programación Dinámica en la práctica y su correlación con la cota teórica calculada en la Sección 4. Para esto, se ejecutan las instancias del dataset dinamica sobre el método DP y se grafican sus resultados en la Figura 6.

Las Figuras 6a y 6b muestran el crecimiento del tiempo de ejecución en función de n y W respectivamente, sobre algunos cortes hechos en la otra variable. Se puede ver que todas las líneas se comportan de manera similar, con un crecimiento lineal en función de ambas variables. Esto se reafirma en la Figura 6c donde se muestra el crecimiento del tiempo de ejecución en función de ambas variables al mismo tiempo. Allí se puede apreciar que el crecimiento es similar tanto en la dirección de n como en W. Finalmente, para confirmar que el tiempo de ejecución de nuestro algoritmo es efectivamente $O(nW)$ como se hipotetiza en la Sección 4, 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 positiva bastante fuerte entre ambas series de datos, lo cuál es confirmado por el índice de correlación de Pearson que es $r \approx 0{,}996075$.

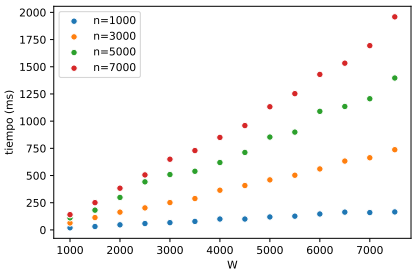
5.7. Experimento 5: Backtracking vs Programación Dinámica

Para finalizar, presentamos un experimento que compara dos técnicas algorítmicas 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 alta, 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 baja programación dinámica debe ser más eficiente.

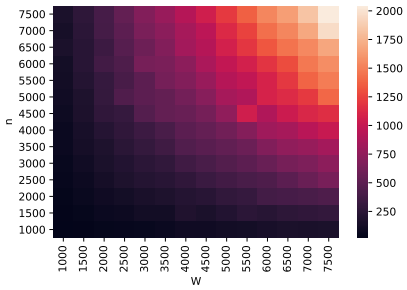
Una observación importante es que ningún algoritmo *domina* al otro en términos de complejidad. Dicho de otro modo, no es cierto que $O(2^n) \subseteq O(nW)$ ni tampoco que $O(nW) \subseteq O(2^n)$.



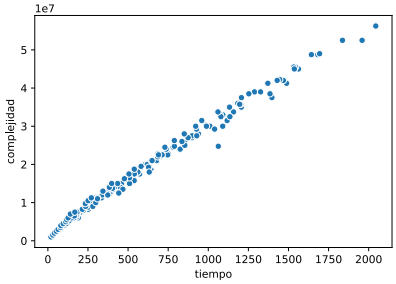
(a) Tiempo de ejecución en función de n.



(b) Tiempo de ejecución en función de W.



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

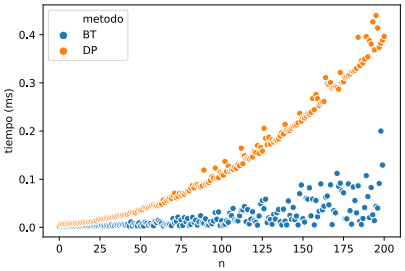


(d) Correlación entre el tiempo de ejeución y la cota de complejidad temporal.

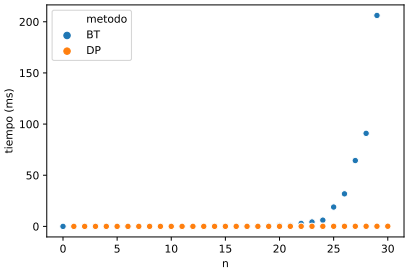
Figura 6: Resultados computacionales para el método DP sobre el dataset dinamica.

Mirando con detenimiento ambas complejidades, podemos observar que el tiempo de ejecución de BT en el peor caso no depende de W, y por lo tanto, ante un valor muy grande el método DP debería degradarse de manera considerable incluso alcanzando un límite de memoria.

La Figura 7 muestra la comparación entre los métodos DP y BT para los datasets densidad-alta y densidad-baja. La hipótesis se confirma, mostrando que DP es más efectivo ante instancias de menor densidad que BT, aunque es robusto para ambos tipos de instancias. Notar que los tiempos de ejecución son bajos en ambos tipos de instancias para DP. El crecimiento de BT en las instancias de densidad baja, sin embargo, es claramente exponencial y hace que en este tipo de instancias la elección segura sea utilizar el algoritmo DP.



(a) Dataset densidad-alta.



(b) Dataset densidad-baja.

Figura 7: Comparación de tiempos de ejecución entre DP y BT.

6. Conclusiones

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

Una línea de trabajo futuro es analizar distintas estructuras de memoización para el método DP de manera tal de poder mitigar el uso de memoria para tamaños grandes de instancias. Esto puede ser combinado con otro tipo de implementación de índole iterativa. Por otra parte, las podas utilizadas en el algoritmo de Backtracking son las más simples para este problema, pero otras reglas más complejas pueden resultar de utilidad en otros contextos.