DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP1 - Knapsack Problem

14 de mayo de 2018 Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Pawlow, Dante	449/12	dante.pawlow@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

http://www.exactas.uba.ar

1. Introducción

En el presente trabajo práctico se estudiará la eficiencia de diversas técnicas algorítmicas (*fuerza bruta*, *meet-in-the-middle*, *backtracking* y *programación dinámica*) en la resolución del problema comúnmente conocido como *problema de la mochila* o *knapsack problem*.
El problema de la mochila toma varias formas, pero en esencia se reduce a maximizar el valor de la carga que se lleva en un contenedor de capacidad limitada. Típicamente se ejemplifica con una mochila (contenedor) en la que se deben llevar artículos con diversos pesos y valores asociados sin superar la capacidad máxima de la mochila.

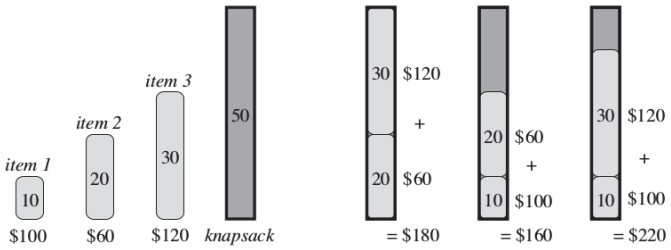

Figura 1: *Ejemplo de problema de la mochila con tres items*

Los algoritmos presentados deberán calcular el máximo valor posible que se puede llevar en una mochila de tamaño W dados los items de peso w_i y valor p_i, pertenecientes al conjunto I.
En la sección *Algoritmos* se detallará el funcionamiento de los algoritmos nombrados, se justificará su correctitud y se analizará su complejidad.
En la sección *Metodología experimental* se detallará cómo se realizaron los experimentos y cuál fue la motivación detrás de cada uno. Luego se analizarán los resultados en la sección *Análisis de resultados*.
Por último, se presentarán las conclusiones del estudio.

Notación

W: capacidad de la mochila
w_i: peso del item i.
p_i: valor del item i.
I: conjunto todos los items.
n: cantidad de items.

2. Algoritmos

2.1. Fuerza Bruta

Los algoritmos de fuerza bruta se caracterizan por buscar todas las soluciones posibles para un problema dado y luego compararlas para encontrar la mejor. Como los espacios de soluciones suelen aumentar considerablemente con el tamaño de entrada, suelen tener complejidades muy altas.
Para generar todas las posibles soluciones a este problema se deben considerar todas las posibles combinaciones de items dentro de la mochila: 2^n. En este caso, esto se logra representando al conjunto de items como array de bits; un 0 representa la ausencia del item en la mochila y un 1, su presencia.
El siguiente algoritmo recorre todos los números naturales de 1 a 2^n, cada uno representando (cuando se lo transforma a binario) una posible combinación de items en la mochila. Luego compara los valores de cada combinación para obtener el máximo.

```
FuerzaBruta(W, I):
    valor_maximo <- 0
    FOR i <- 1 TO 2^n DO:
        c <- i
        valor, carga, posicion_bit <- 0
        WHILE c > 0 DO:
            IF c es impar DO:
                valor + valor de I[posicion_bit]
                carga + tamano de I[posicion_bit]
            END IF
            posicion_bit + 1
            se desplazan los bits de c una posicion a la derecha
        END WHILE
        IF entra en la mochila y su valor es mayor al maximo DO:
            valor_maximo <- valor
        END IF
    END FOR
    RETURN valor_maximo
```

2.1.1. Correctitud

La correctitud de este algoritmo es relativamente sencilla de demostrar, ya que genera todas las combinaciones de items, suma los valores de cada una y compara la suma de valores para encontrar la combinación que produce la mochila de valor máximo.
Es importante ver que con este algoritmo se generan todas las combinaciones. Esto se puede deducir de que genera todos los números binarios que se pueden representar con n bits.

2.1.2. Complejidad

Este algoritmo recorre desde 1 hasta 2^n, lo que implica una complejidad de al menos $\mathcal{O}(2^n)$. Además, por cada iteración i recorre todos los bits de ese número; como i puede ser hasta 2^n, puede llegar a tener n bits. Entonces, como el resto de las operaciones tienen una complejidad $\mathcal{O}(1)$, resulta en una complejidad total de $\mathcal{O}(n \times 2^n)$.

2.2. Meet in the middle

Este algoritmo divide a la lista de items de entrada en dos partes. Luego, busca por fuerza bruta todas las combinaciones posibles dentro de cada sublista para una mochila de tamaño W.
En una segunda parte busca la mejor combinación entre los dos subconjuntos de soluciones resultantes. Para hacerlo con mejor complejidad temporal, ordena ambos subconjuntos crecientemente por peso total de las combinaciones de items; esto permite hacer búsqueda binaria sobre los elementos.
Queda, por último, el caso en el que la mochila de mayor valor no es la de peso máximo. Para esto, a cada item se le asigna como valor el máximo entre el suyo y el del item de mayor valor más pequeño que él.

```
MeetInTheMiddle(W, I):
    I1 <- primera mitad de los elementos de I
    I2 <- segunda mitad de los elementos de I

    1) Se aplica fuerza bruta a ambos I1 e I2 y se
       guardan las combinaciones en mochilas1, mochilas2.
    2) Se ordenan por peso creciente mochilas1 y mochilas2.
    3) Se guardan solamente las combinaciones con mayor
       valor para cada peso.
    4) Se recorren linealmente (por separado) mochilas1 y 2.
       Si el valor de la mochila i es menor que el de la i-1,
       se reemplaza su valor original por el de la i-1.
    5) Se recorre linealmente mochilas1, se busca binariamente
       sobre mochilas2 para una con peso (W-mochilas1[i]).
    6) Se retorna el mayor valor encontrado en (5).
```

2.2.1. Correctitud

Este algoritmo separa I en dos subconjuntos y busca todas las combinaciones posibles (se puede pensar como conjunto de partes de cada uno de los subconjuntos). Luego se realiza el equivalente al producto cartesiano entre ambos subconjuntos (pero de una manera más eficiente), lo que produce todas las posibilidades de I. Finalmente, se encuentra el valor máximo entre todas las combinaciones.

2.2.2. Complejidad

Se aplica fuerza bruta sobre ambas mitades de I, resultando en $\mathcal{O}(2 \times 2^{\frac{n}{2}})$. Por otra parte se ordenan los subconjuntos ($\mathcal{O}(2 \times \log 2^{\frac{n}{2}} \times 2^{\frac{n}{2}}) = \mathcal{O}(n \times 2^{\frac{n}{2}})$), luego se *filtran* recorriéndolos linealmente ($\mathcal{O}(2 \times 2^{\frac{n}{2}})$) y por último se comparan.
La comparación se realiza recorriendo linealmente el primer subconjunto y buscando binariamente un único elemento en el segundo por cada iteración. Esto resulta en una complejidad de $\mathcal{O}(2^{\frac{n}{2}} \times \log 2^{\frac{n}{2}}) = \mathcal{O}(n \times 2^{\frac{n}{2}})$
Entonces, como todas esas operaciones son independientes, la complejidad resultante es $\mathcal{O}(2 \times 2^{\frac{n}{2}} + n \times 2^{\frac{n}{2}} + 2 \times 2^{\frac{n}{2}} + n \times 2^{\frac{n}{2}}) = \mathcal{O}(n \times 2^{\frac{n}{2}})$.

2.3. Backtracking

Los algoritmos de backtracking se caracterizan por explorar todas las posibles soluciones de un problema. Se puede pensar el espacio de búsqueda como un árbol, siendo una hoja una posible solución al final de una rama de ejecución. A diferencia de los algoritmos de fuerza bruta, los de backtracking van verificando las soluciones mientras las construyen; esto permite descartar ramas de ejecución que pueden llevar a una solución inválida o subóptima, evitando así una costosa verificación final.

A los algoritmos de backtracking se les pueden aplicar *podas*. Una poda es una función que predice si una rama de ejecución (válida hasta el momento) producirá una solución inválida o subóptima y cortar su ejecución con mayor anticipación.
Existen dos tipos de poda: de *factibilidad* y de *optimalidad*. Como su nombre lo indica, las de factibilidad actúan si las soluciones derivadas de la rama de ejecución resultarán inválidas, mientras que las de optimalidad lo hacen si serán subóptimas respecto a la mejor solución encontrada hasta el momento.
A pesar de que permiten reducir las ramas procesadas, las podas pueden aumentar la complejidad del algoritmo o causar tiempos de ejecución más altos cuando son funciones con complejidades no triviales.

```
Backtracking(i, mochila, I):
    IF no quedan items OR podaFactibilidad(i, mochila, I)
       OR podaOptimalidad(i, mochila, I) DO:
        RETURN mochila
    END IF

    mochila_sin_item <- Backtracking(i + 1, mochila, I)
    IF la mochila con item i se pasa de peso DO:
        RETURN mochila_sin_item
    END IF

    -> Se agrega el siguiente item a la mochila
    mochila_con_item <- Backtracking(i + 1, mochila, I)
```

```
IF vale mas la mochila con item DO:
    RETURN mochila_con_item
ELSE DO:
    RETURN mochila_sin_item
END IF
```

2.3.1. Correctitud: Backtracking puro

En principio, como busca todas las soluciones posibles, está garantizado que la encuentra (si existe). La diferencia fundamental con *fuerza bruta* es que si llega a una solución parcial inválida, no sigue explorando esa rama. Las soluciones parciales son inválidas si la suma de los pesos de los items sobrepasa la capacidad de la mochila.
Se puede observar que compara el valor de las mochilas que encuentra y devuelve siempre la de mayor valor. Como es un algoritmo recursivo y todas las llamadas devuelven la mochila de mayor valor, el resultado final va a ser una solución óptima.

2.3.2. Complejidad: Backtracking puro

Es relativamente complicado encontrar la complejidad de un algoritmo recursivo respecto de la de uno imperativo. Primero hay que calcular la cantidad máxima de llamadas recursivas que puede hacer y luego considerar la complejidad agregada por cada una de ellas. En este caso, puede haber un máximo de 2^n llamadas recursivas (en el caso en el que se recorren todas las ramas) y todas las operaciones internas tienen complejidad $\mathcal{O}(1)$. Por lo tanto, la complejidad del algoritmo es de $\mathcal{O}(2^n)$.

2.3.3. Correctitud: poda de factibilidad

La poda de factibilidad verifica si todavía quedan elementos que quepan en la mochila en una rama de ejecución dada. Esto se logra buscando linealmente el item de menor tamaño entre los restantes y verificando que cabrá.

2.3.4. Complejidad: poda de factibilidad

Es una búsqueda lineal con comparaciones $\mathcal{O}(1)$ (verifican una propiedad numérica del objeto item) que agrega una complejidad de $\mathcal{O}(n)$ a la de backtracking puro, siendo entonces $\mathcal{O}(n \times 2^n)$ la complejidad de backtracking con la poda.

2.3.5. Correctitud: poda de optimalidad

La poda de optimalidad debe poder predecir, para una solución parcial dada, si existe al menos una rama hija cuya solución sea mejor que la mejor solución encontrada hasta el momento. Esto permite podar las ramas que no vayan a tener una solución óptima.
Es importante que la predicción sea una cota superior del máximo alcanzable por las ramas hijas, de lo contrario se podría podar erróneamente una rama. A su vez, se usa una aproximación porque encontrar el valor exacto sería tan costoso como simplemente resolver el problema.
Una buena cota superior proviene del *problema de la mochila fraccional*, en el que se pueden fraccionar los items para llenar completamente la capacidad de la mochila.

```
problemaDeLaMochilaFraccional(mochila, I):
    //Precondicion: los items deben estar ordenados
    //decrecientemente por el cociente valor/peso
    FOR item de I DO:
        IF el item entra en la mochila DO:
            agregar item a la mochila.
        ELSE DO:
            agregar la fraccion del item que entre, agregando
            la fraccion correspondiente de su valor a la mochila.
            RETURN
        END IF
    END FOR
    RETURN
```

Para reducir la complejidad de la poda, se ordenan los items de forma decreciente por el cociente de su valor y su peso. Esto se hace antes de comenzar a recorrer las soluciones con backtracking. Como I es un conjunto, cualquier ordenamiento es equivalente.

2.3.6. Complejidad: poda de optimalidad

De manera similar a la poda de factibilidad, recorre linealmente los items restantes, así que agrega una complejidad de $\mathcal{O}(n)$ al backtracking puro, resultando en una complejidad total de $\mathcal{O}(n \times 2^n)$.
El ordenamiento se produce con una complejidad de $\mathcal{O}(n \times \log n)$, pero se produce antes de comenzar a explorar las soluciones, así que es independiente del backtracking propiamente dicho.

2.4. Programación dinámica

Los algoritmos de programación dinámica resuelven problemas combinando las soluciones de subproblemas. A diferencia de la técnica *divide and conquer*, la técnica de programación dinámica requiere que haya solapado de problemas, ya que una de sus características fundamentales es almacenar la solución a los subproblemas ya calculados para evitar calcularlos nuevamente.
Los problemas que se pueden resolver por programación dinámica deben, a su vez, cumplir el *principio de optimalidad de Bellman*: soluciones óptimas de un problema incorporan soluciones óptimas de los subproblemas asociados, que se pueden calcular independientemente.

Existen dos métodos para escribir un algoritmo de programación dinámica: *top down* y *bottom up*. El método *top down* comienza con el problema que se quiere resolver y va resolviendo recursivamente los subproblemas asociados. El método *bottom up* (el utilizado en este trabajo práctico) comienza resolviendo los subproblemas de menor tamaño y progresa iterativamente hasta llegar al problema original.

El algoritmo utilizado implementa el método *bottom up*, recorriendo una matriz $A^{(W+1)\times(n+1)}$. La función que se quiere calcular para resolver el problema es la siguiente:

$f(i, w) =$ Máximo valor para una mochila de peso W con los items $\{1, 2, ..., i\}$

```
ProgramacionDinamica(W, I):
    se genera el cache (matriz de (W+1)*(n+1))
    FOR size <- 1 TO W+1 DO:
        FOR i <- 1 TO #I DO:
            IF tamano(I[i-1]) > size DO:
                cache[size][i] = misma solucion al anterior
            ELSE DO:
                cache[size][i] = el mayor valor entre agregar
                el item o no
            END IF
        END FOR
    END FOR
    RETURN cache[W][n]
```

2.4.1. Correctitud

Al ir recorriendo los subproblemas necesarios para construir la solución, el algoritmo se asegura obtener todas las soluciones óptimas a subproblemas con las que construye la solución óptima al problema original.
El problema cumple el principio de optimalidad de Bellman: considérese el item más valioso que pesa como máximo w; si se quita el item j, la carga restante

debe ser la más valiosa que pese como mucho $W - w_j$ que se pueda tomar de $I - 1$.

2.4.2. Complejidad

Una de las ventajas de la técnica *bottom up* respecto de la *top down* es que la complejidad es más sencilla de calcular. En este caso resulta evidente que se trata de una complejidad $\mathcal{O}(n \times W)$, ya que hay dos *for* anidados y las operaciones que se realizan dentro de ellos son $\mathcal{O}(1)$.

3. Método experimental y análisis de resultados

El objetivo de este trabajo práctico es comparar los tiempos de ejecución de las diversas técnicas algorítmicas y ver cómo se comportan respecto a su complejidad teórica, para lo cual se diseñaron diversos experimentos.

Algo importante a tener en cuenta al momento de compararlos es la varianza producida tanto por las instancias como por factores externos a la experimentación relacionados con procesos del sistema operativo y otros tomando recursos de manera variable. Es por esto que se decidió repetir 50 veces las mediciones para cada n en todos los casos, lo que permite reducir la varianza. Esto tiene su sostén teórico en la *Ley de los grandes números*, que establece que el promedio de los resultados de un proceso aleatorio converge a la esperanza de dicho proceso en función de la cantidad de reiteraciones.

3.1. Comparando todas las técnicas algorítmicas

En la *figura 2* se presentan las cuatro técnicas algorítmicas utilizadas. Los gráficos representan los experimentos que se pensaron fijando o variando las diversas variables disponibles, para explorar el comportamiento de los algoritmos en estos casos.

- *Gráfico 2a*: $W = n$; $\forall i, w_i \leq \frac{W}{2}$, con p_i aleatorio.
- *Gráfico 2b*: $W = n$; $\forall i, w_i = 1$, con p_i aleatorio.
- *Gráfico 2c*: $W = n$; $\forall i, w_i = 1 \wedge p_i = 1$.
- *Gráfico 2d*: $W = 10000$; $\forall i, w_i \leq \frac{W}{2}$, con p_i aleatorio.

Las diferencias entre fuerza bruta y meet in the middle por un lado y backtracking y programación dinámica por otro fueron tan grandes que sólo se pueden apreciar las curvas en escala logarítmica. Esto lleva a pensar que la forma más precisa de estudiar estos algoritmos es por separado.

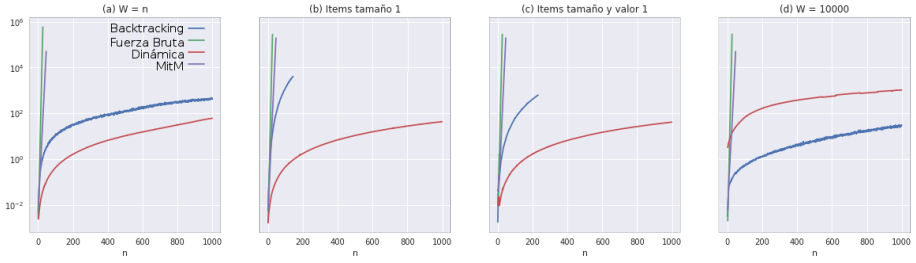

Figura 2: *Comparación de desempeño, todos los algoritmos. Escala logarítmica*

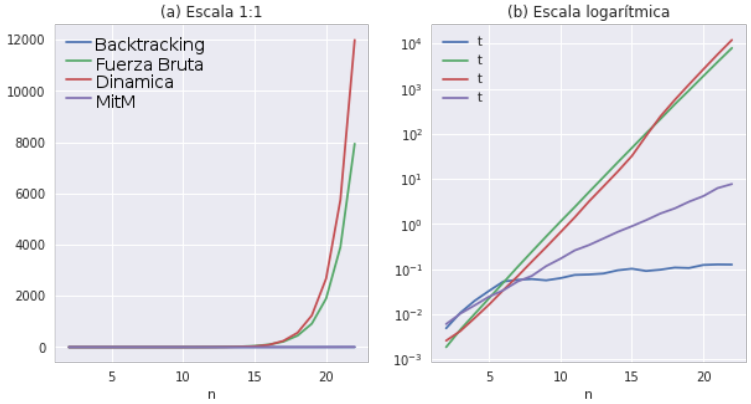

Figura 3: *Gráfico para W exponencial respecto de n*

El *gráfico 2d* fue ideado para evaluar si existe alguna instancia en la que programación dinámica tarde tanto o más que los otros algoritmos. Pero dado que su complejidad es $\mathcal{O}(n \times W)$, fijar W en una constante (sin importar cuán alta), sólo puede aumentar tiempos de manera fija, no cambiar la pendiente de crecimiento.
Siguiendo esta idea, se planteó otro experimento, que se puede apreciar en la *figura 3*. En él, W crece exponencialmente respecto de n. Esto lleva a un comportamiento exponencial, como se observa en el *gráfico 3b*, que presenta los mismos datos en escala logarítmica.

Pese a este caso en el que presentan comportamientos relativamente similares, en general estos algoritmos tienen comportamientos considerablemente diferentes (como se aprecia en la *figura 2*). Por este motivo, en las siguientes secciones se analizan por separado.

3.1.1. Fuerza Bruta y Meet in the Middle

En la *figura 4* se exponen los algoritmos de fuerza bruta y meet in the middle, utilizando los mismos tipos de instancia que en la *figura 2*.
Se puede observar que estos algoritmos tienen comportamiento exponencial (son lineales en escala logarítmica), como se predijo teóricamente. También se observa en la escala logarítmica que las pendientes de las rectas de meet in the middle son aproximadamente la mitad que las de fuerza bruta, lo que se condice con la diferencia entre las complejidades teóricas de ambos algoritmos.

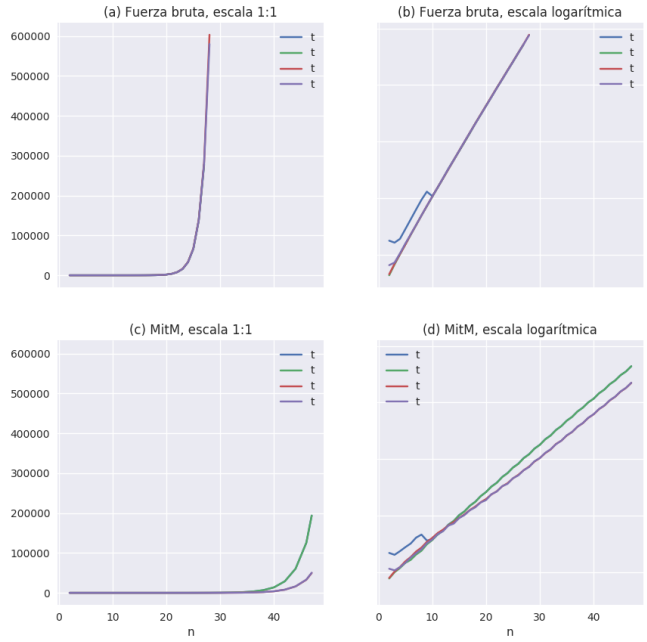

Figura 4: *Comparación de desempeño, algoritmos exponenciales*

Otra observación interesante es que el desempeño del algoritmo de fuerza bruta no varía significativamente con las distintas instancias como sí lo hace el resto. Esto parece ser así por su forma de generar y recorrer las subsoluciones del problema, que es independiente de las características de las instancias particulares (porque recorre siempre todas).

3.1.2. Backtracking y podas

En la *figura 5* se presentan los algoritmos de backtracking puro, con poda de factibilidad, de optimalidad y ambas combinadas. Se utilizan las mismas instancias que para la *figura 2*.

Se observa algo curioso: la poda de optimalidad parece reducir considerablemente los tiempos de ejecución, pero no se puede establecer que siga una cota definida. Esto tiene sentido porque la cota se beneficia de la resolución lineal del *fractional backpack problem* para reducir drásticamente las ramas exploradas, pero sigue teniendo complejidad exponencial en peor caso.
La poda de factibilidad no presenta tanta eficacia y se asemeja más a la complejidad teórica esperada.

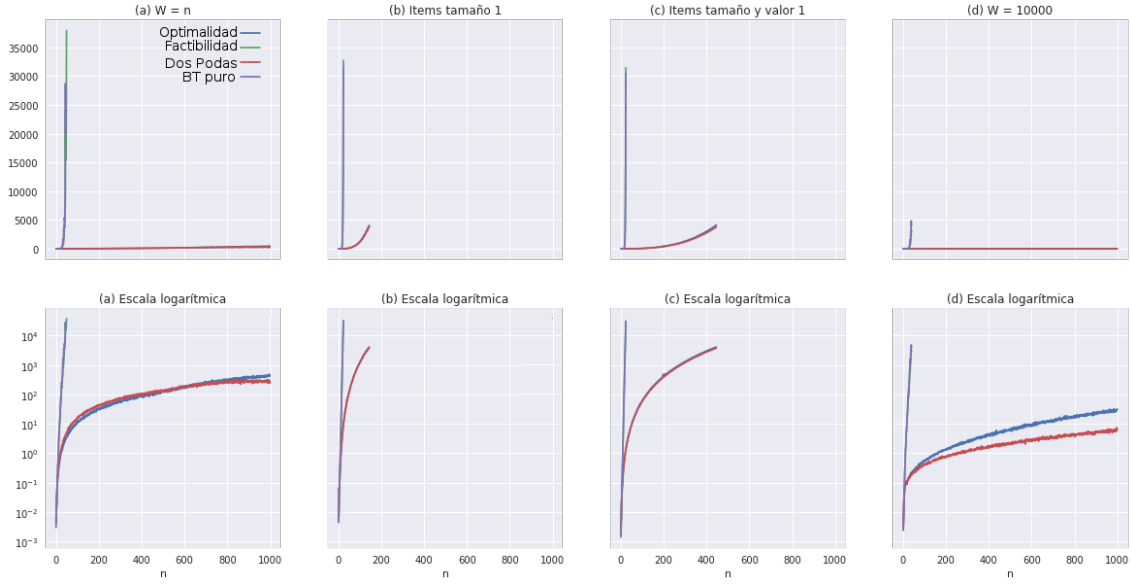

Figura 5: *Comparación de desempeño, backtracking puro y con podas.*

3.1.3. Programación Dinámica

En la *figura 6* se pueden apreciar las diversas formas que puede tomar la ejecución del algoritmo de programación dinámica. Al contrario que el resto de los algoritmos, el crecimiento depende tanto de W como de n, lo que lleva a que pueda variar radicalmente para los mismos rangos de n, dependiendo de cómo crezca W. Cuando W se mantiene constante, adopta una apariencia lineal respecto de n. Cuando W crece a la par de n, toma forma cuadrática. Incluso, como se vio en la *figura 3*, existen circunstancias en las que puede adoptar comportamientos exponenciales, por ejemplo: cuando $W = 2^n$.

Figura 6: *Tiempos de ejecución de programación dinámica para diversas instancias*

4. Conclusiones

En este trabajo práctico se buscaba evaluar el desempeño de los cuatro algoritmos en relación con sus complejidades temporales y entre ellos.

Una de las conclusiones que se pueden extraer es que a veces la complejidad teórica y el desempeño práctico pueden estar considerablemente disociados. Esto se observó muy claramente para backtracking y la poda de optimalidad, que en los casos estudiados se desenvolvieron de forma considerablemente mejor que la esperada por su complejidad.

Otra conclusión importante es que los algoritmos presentan comportamientos distintos ante instancias de distintas características. Esto produce que pueda ser conveniente elegir el algoritmo más apropiado si tenemos alguna información sobre la instancia a resolver. Como ejemplo: no elegiríamos programación dinámica en el caso en el que W crece exponencialmente respecto de n.