DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP1 - Knapsack Problem

16 de abril 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

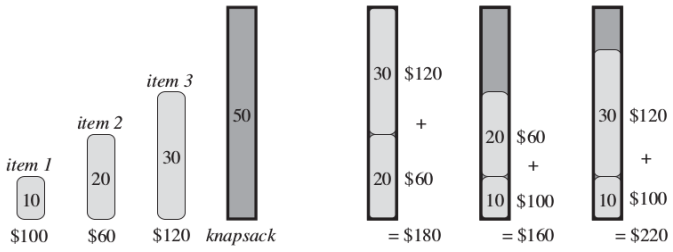
Ciudad Autónoma de Buenos Aires - Rep. Argentina

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.



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 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 0 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 es una variante del algoritmo de fuerza bruta. La diferencia radica en que la entrada se divide en dos y la respuesta se construye buscando las mejores respuestas compatibles de ambas partes. Esto permite resolver entradas de mayor tamaño que el algoritmo de fuerza bruta no puede tratar en tiempos razonables. El problema es que en muchos casos la verificación final agrega un costo importante y es la principal causa por la cual no se puede dividir la entrada en tres o más.
El algoritmo desarrollado divide la lista de items en dos y utiliza una variante del algoritmo de fuerza bruta.

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

    mochilas1 = VarianteFuerzaBruta(I1) //Devuelve un vector de tuplas (valor, p
    mochilas2 = VarianteFuerzaBruta(I2)

    valor_maximo <- 0
    FOR cada tupla m1 en mochilas1 DO:
        FOR cada tupla m2 en mochilas2 DO:
            IF carga(m1) + carga(m2) < W AND valor(m1) + valor(m2) > valor_maxim
                valor_maximo <- valor(m1) + valor(m2)
            END IF
        END FOR
    END FOR
    RETURN valor_maximo
```

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, 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 comparan los subconjuntos resultantes, operación que se desarrolla con complejidad $\mathcal{O}(2^{\frac{n}{2}} \times 2^{\frac{n}{2}}) = \mathcal{O}(2^n)$. Entonces, la complejidad resultante es $\mathcal{O}(2 \times 2^{\frac{n}{2}} + 2^n) = \mathcal{O}(2^n)$.

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. Se trata, entonces, de verificaciones que permiten ahorrar tiempo de 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 en algunos casos causar tiempos de ejecución más altos cuando son funciones con complejidades no triviales.

Primero se analizará el algoritmo de backtracking sin podas ("puro") y luego las podas de factibilidad y optimalidad, con sus respectivas complejidades.

```
Backtracking(i, mochila, I):
    IF no quedan items 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

De forma similar a como trabaja fuerza bruta, backtracking busca todas las soluciones posibles, con la diferencia de que explora el árbol de manera recursiva. En principio, como busca todas las soluciones posibles, está garantizado que la encuentra (si existe).
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.
El algoritmo realiza una verificación para evitar explorar ramas inválidas: verifica si a agregar el item correspondiente no se sobrepasa la capacidad de la mochila.

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. Para facilitar este proceso, se ordenan los elementos de manera creciente por peso antes de hacer la primera llamada recursiva. Esto permite hacer la verificación simplemente consultando el peso del primer elemento de los que quedan por verificar.

2.3.4. Complejidad, poda de factibilidad

Verificar la propiedad peso del primer elemento de un vector de items se realiza en $\mathcal{O}(1)$.

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.

```
DynamicProgramming(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.

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 lograr este objetivo se diseñaron diversos experimentos con instancias aleatorias en donde se midieron estos tiempos.

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.
Para obtener una representación más significativa, se repitieron 50 veces los experimentos por algoritmo para n. Por la *ley de los grandes números*, tomar el promedio de repeticiones mayores a 30 nos permite asegurarnos de que la varianza para cada *datapoint* se mantiene dentro de los márgenes aceptables.

3.1. Comparando todas las técnicas algorítmicas

En la *figura 1* se representaron las cuatro técnicas algorítmicas utilizadas. Inmediatamente se puede observar la diferencia entre programación dinámica y las otras técnicas algorítmicas: es tan considerable que a simple vista parece constante.

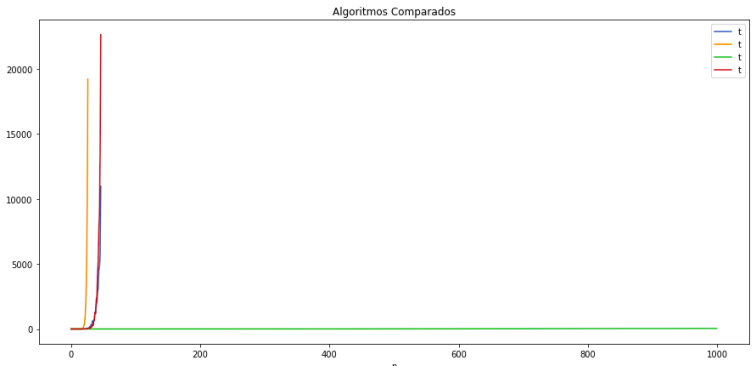


Figura 1: *Comparación de desempeño, todos los algoritmos*

En la *figura 2* se grafica la misma tabla con escala logarítmica en el eje y (que representa tiempo). Se puede observar, efectivamente, que la programación dinámica no produce un gráfico constante.
La diferencia es tan considerable que, para estudiarlos, se separarán los algoritmos en dos grupos: exponenciales (fuerza bruta, meet in the middle y backtracking) y no exponenciales (programación dinámica).

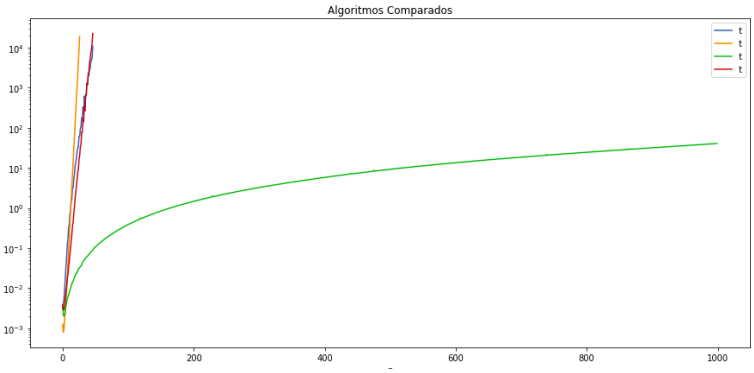


Figura 2: *Comparación de desempeño, todos los algoritmos con escala logarítmica*

3.1.1. Algoritmos exponenciales

En la *figura 3* se exponen los algoritmos teóricamente exponenciales. Lo primero que se observa es que los tiempos del algoritmo de fuerza bruta escalan considerablemente más rápido. Esto es de esperar porque, por un lado, meet in the middle tiene una complejidad comparativamente menor y, por otro, porque es de esperar que las podas que ocurren en backtracking reduzcan lo tiempos de ejecución.

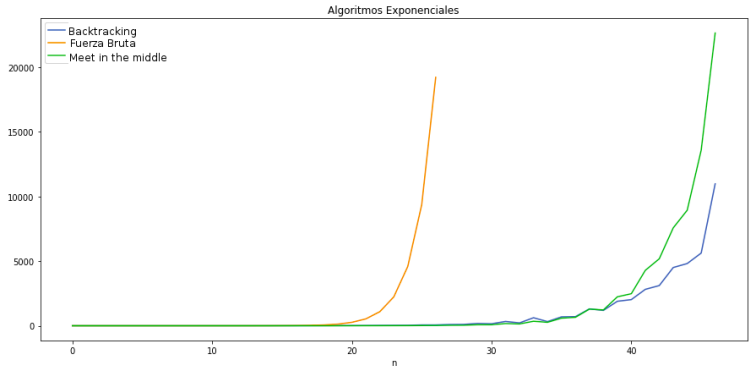


Figura 3: *Comparación de desempeño, algoritmos exponenciales*

La *figura 4* muestra los mismos algoritmos en escala logarítmica, en donde se puede observar una tendencia lineal. Esto demuestra que se trata de funciones exponenciales.

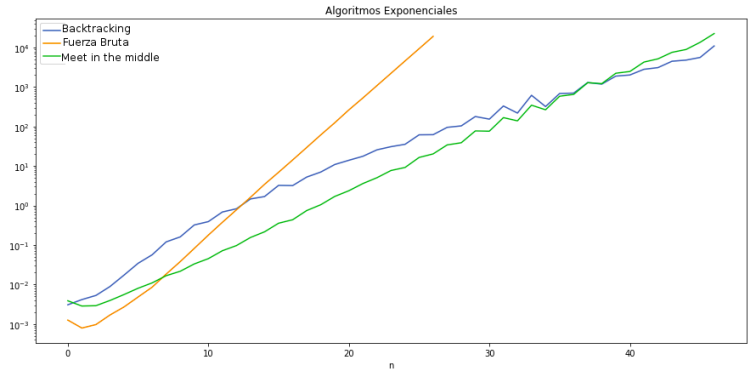


Figura 4: *Comparación de desempeño, algoritmos exponenciales con escala logarítmica*

3.1.2. Backtracking y podas

En el *gráfico 5* se observa el desempeño de backtracking en sus diversos estados: sin poda, con poda de factibilidad, con poda de optimalidad y con ambas podas. Sorprendentemente, el algoritmo que presenta los peores tiempos de ejecución es el de poda de optimalidad. Esto, en conjunto con el hecho de que no hay diferencia significativa entre el algoritmo con poda de factibilidad y el que tiene ambas podas, sugiere que la poda de optimalidad no presenta un beneficio apreciable. Por el contrario, parece aumentar los tiempos sin podar significativamente.

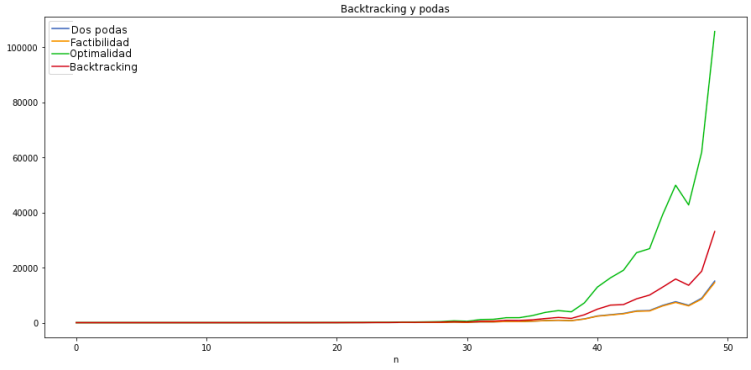


Figura 5: *Comparación de desempeño, backtracking puro y con podas.*

Por otra parte, la poda de factibilidad presenta un excelente desempeño, mejorando apreciablemente los tiempos respecto al backtracking puro.

3.1.3. Programación Dinámica

Para este algoritmo, el experimento se realizó incrementando n y W al mismo ritmo. Analizando la *figura 6* se puede observar que esto produjo que los tiempos de ejecución crecieran polinomialmente, como era de esperarse.

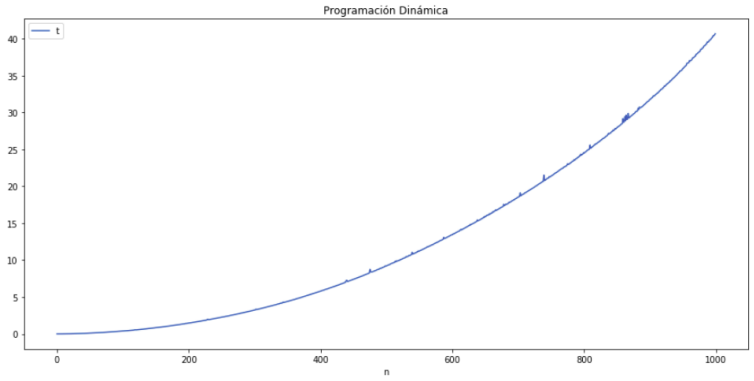


Figura 6: *Comparación de desempeño, backtracking puro y con podas.*

4. Conclusiones

Nota al corrector: Por razones de último momento no pude terminar con algunos detalles como los ajustes a exponenciales/polinómicas de los gráficos y algunos otros detalles. Por esta razón me parece que no sería correcto sacar conclusiones sin esos datos.
Entrego igual para poder tener un feedback sobre lo que hice hasta ahora, que creo que es la mayoría del TP.

Gracias por comprender :) (y disculpas por la molestia!)

Dante Pawlow