DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Reentrega Trabajo Práctico Nº1

Dame la mochila!

13 de mayo de 2018 — Algoritmos y Estructuras de Datos 3

Integrante	LU	Correo electrónico
Joaquin Romera	183/16	joakromera@gmail.com

Instancia	Docente	Nota
Primera entrega		

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

Índice

1. Introducción

1.1. Knapsack Problem

El objetivo del presente trabajo es implementar y probar algoritmos que resuelvan el conocido problema de la mochila (Knapsack Problem). En el mismo buscamos un subconjunto de ítems, con pesos y beneficios determinados, que maximice el beneficio total sin excedernos de un peso límite. O sea:

$$maximizar \sum_{j=1}^{n} p_j x_j$$

$$sujeto\ a \sum_{j=1}^{n} w_j x_j \leq W\ ,\ x_j \in \{0,1\}, j = 1, ..., n$$

La importancia del problema de la mochila radica en que puede modelar innumerables situaciones desde simples elecciones cotidianas a procesos complejos de tomas de decisiones. Si bien el problema es intrínsicamente sencillo, el crecimiento combinatorio al considerar números cada vez mayores de elementos hace que su resolución se vuelva impracticable por algoritmos que surjan de una intuición muy básica. En los algoritmos propuestos exploramos opciones para encontrar una solución óptima al problema partiendo desde lo más intuitivo a soluciones más eficientes. Finalmente compararemos su desempeño cotejando que se respete la complejidad esperada y analizaremos algunos casos borde y posibles optimizaciones.

1.2. Algoritmos propuestos

Consideraremos tres algoritmos que permiten encontrar soluciones a distintos knapsack problems: un algoritmo por fuerza bruta donde exploramos todas las combinaciones de ítems posibles (sin importar si una combinación particular está dentro de las restricciones o no) y luego elegimos la de mayor beneficio que no exceda W (el límite o peso máximo), una versión de backtracking donde calculamos los beneficios de todos los subconjuntos que respeten el límite de la mochila (veremos que en ciertos casos su comportamiento es semejante al anterior aunque en promedio su rendimiento es mejor), y finalmente una implementación según la técnica de programación dinámica en la cual guardamos resultados parciales para ahorrar tiempo de cómputo en cálculos subsiguientes. En cada uno ofrecemos una descripción acompañada del pseudocódigo correspondiente, la correctitud y el cálculo de su complejidad.

Algunas definiciones útiles para la descripción de los algoritmos y sus pseudocódigos:

- n = cantidad de ítems
- W = peso máximo o límite de la mochila
- $items$ = lista de n ítems
- Un *ítem* es una tupla tal que el primer componente es el peso y el segundo su beneficio.

2. Algoritmos

2.1. Brute Force

2.1.1. Descripción

Como el nombre lo indica, este algoritmo busca una solución evaluando todas las posibles combinaciones de ítems y eligiendo la de mayor beneficio que no exceda el peso límite de la mochila. Encontrar todas las combinaciones equivale a calcular el conjunto de partes de un conjunto, si tenemos n ítems de entrada el cardinal del conjunto de partes de la entrada es 2^n. Luego para quedarnos con la mejor solución deberíamos calcular la suma de cada uno de los subconjuntos y elegir la mayor -cuidando que el peso respete la restricción dada-. Si calculamos esta suma al momento de elegir si un elemento va en un subconjunto o no, después de obtener el conjunto de partes iteramos sobre este buscando un máximo posible. Más aún, si además guardamos una solución temporal y solo la reemplazamos cuando obtenemos un nuevo subconjunto y la suma parcial de éste es mayor, al terminar de calcular partes de la entrada tenemos la solución final en esta variable. Esta idea es implementada recursivamente en el algoritmo propuesto.

2.1.2. Pseudocódigo

Partial Sums recorre toda la lista de ítems incrementando el índice en cada iteración y llamándose recursivamente a sí mismo. Por cada invocación del proceso se genera un *index* y un *par* para llevar registro de qué ítems faltan evaluar y para almacenar la suma de beneficios y pesos actuales. Las llamadas recursivas contemplan agregar o no agregar cada ítem. El caso base es cuando se llega al final de la lista, donde se evalúa si el peso del *par* actual es menor que w y si su beneficio es mayor al de *solution*. De ser afirmativo la reemplaza, luego finaliza. Al terminar el proceso tenemos la solución en *solution*.

Algorithm 1. Brute Force

```
1: procedure BRUTE FORCE(int W, lista(tuplas) items)
2:     par ← (0, 0)
3:     solution ← (0, 0)
4:     index ← 0
5:     partialSums(index, par, solution, W, items)        ▷ Solution e Items pasan por referencia
6:     return solution.second
```

Algorithm 2. Partial Sums

```
1: procedure PARTIALSUMS(int index, par, &solution, W; lista(tuplas) &items)
2:     if index = |items| then
3:         if (par.first ≤ W) ∧ (par.second > solution.second) then
4:             solution.first ← par.first
5:             solution.second ← par.second
6:     else
7:         index ← index + 1
8:         partialSums(index, par, solution, W, items)        ▷ No agregamos ítem
9:         par.first ← par.first + items[index − 1].first
10:        par.second ← par.second + items[index − 1].second
11:        partialSums(index, par, solution, W, items)        ▷ Agregamos ítem
```

2.1.3. Correctitud

Primero veamos que vale $P(n)$ = el algoritmo termina cuando $index = |items| = n$, $\forall n \geq 1$. Usamos sin demostración que al término de la n-ésima llamada recursiva $index = n$. El caso para $P(1)$: al comienzo del algoritmo $index \leftarrow 0$, el subproceso PartialSums incrementa $index$ en 1 (ver línea 7 del pseudocódigo) y hace dos llamadas recursivas (líneas 8 y 11). Ahora $index = |ítems| = 1$ y por cumplir la condición de la línea 2 no realiza más llamadas y termina. Suponiendo que vale la hipótesis para $P(n)$, probemos que vale para $P(n+1)$. Al comienzo de la n-ésima llamada a PartialSums $index = n$, nuevamente se incrementa $index$ en 1 por línea 7 y hace dos recursiones más. En estas ahora $index = n+1 = |ítems|$ y por el mismo argumento el algoritmo termina en la línea 2. Luego, vale $P(n)$, $\forall n \geq 1$.

Ahora queremos probar que el algoritmo devuelve la mayor suma de todas las combinaciones posibles de elementos sujeta al límite de peso W. Sea $P(n)$ = al terminar el algoritmo en *solution* está el mayor beneficio de todas las combinaciones posibles sujetas a W. Para $P(1)$ al principio de PartialSums tenemos los valores: $solution = (0,0)$, $par = (0,0)$ e $index = 0$. Luego se hacen dos llamadas recursivas con $index = 1$ en las líneas 8 y 11. En la primera los valores de *solution* y *par* no se modifican, en la segunda se suman los valores del i-ésimo ítem (líneas 9 y 10) -notamos que ambas agotan todas las combinaciones posibles para $|items| = 1$-. En las nuevas llamadas $index = 1 = |items|$ por lo que se procede a comparar los pesos con W y los beneficios con *solution*. Sea par_0 los valores de peso y beneficio acumulados sin agregar el ítem y sea par_1 agregándolos, en las líneas 3 a 5 como $par_0 = (0,0)$ no se cumple la guarda del condicional ya que no vale $par_0.second > solution.second$, luego en este caso el beneficio máximo de no considerar ningún ítem es 0. Para par_1 tenemos dos posibilidades: si $par_1.first > W$ el algoritmo vale ya que al no poder considerar ninguno de los ítems para el límite W, el beneficio es 0; en cambio si $par_1.first \leq W$ vale la guarda ya que estamos considerando solo casos no triviales (pesos y beneficios positivos) por lo que $par_1.second > solution.second = 0$. Finalmente tenemos en *solution* el mayor beneficio posible para $P(1)$.

Probamos ahora el caso para $P(n+1)$ considerando que vale para $P(n)$. Por hipótesis inductiva tenemos en *solution* el máximo beneficio de todas las combinaciones posibles sujetos a W, estamos en una rama del árbol de recursión donde *par* tiene acumulados los pesos y beneficios de los ítems agregados por la rama y estamos considerando el caso del $n+1$-ésimo ítem. Por la línea 7 $index = |items| = n+1$ y se realizan dos llamadas sumando y no sumando el ítem actual. En estas se procede a evaluar la condición de la línea 3 y tenemos los mismos casos que en el caso para $P(1)$. Sean par_0 y par_1 siguiendo el criterio anterior, como vale la hipótesis inductiva para $P(n)$ tenemos en *solution* el máximo beneficio. Si los pesos de par_i exceden W no se modifica *solution*, si los pesos no se pasan del límite entonces se evalúa si el beneficio actual del par_i es mayor al de *solution* y la actualizan de ser afirmativo. Como el orden no influye en la comparación si un par_i tiene mejor resultado nos queda $solution = par_i$, si ambos son mejores $solution = max(par_0, par_1)$. Luego vale $P(n)$ ya que el algoritmo devuelve el valor acumulado en *solution* y es la mayor suma de todas las combinaciones posibles de elementos sujeta al límite de peso W.

2.1.4. Complejidad

En el caso de que la lista tenga un solo elemento tenemos:

$$T(1) = T(0)$$

Para los $n > 1$ cada llamada produce dos de tamaño $n-1$, luego:

$$T(2) = T(1) + T(1) = 2T(0)$$
$$T(3) = T(2) + T(1) + T(0) = 2T(2)$$
$$T(N) = 2T(N-1) = 4T(N-2) = ... = 2^{N-1}T(1)$$

Lo que nos da: $O(2^N)$

2.2. Backtracking

2.2.1. Descripción

A diferencia del algoritmo de fuerza bruta, el backtracking evalúa si tiene sentido explorar determinados casos (en términos de que satisfagan las restricciones de W), deteniendo el cálculo de una rama cuando una solución parcial no puede progresar a una solución válida. Esta *poda por factibilidad* se da cuando la suma parcial actual se excede del límite agregando el ítem siguiente, o cuando la suma de los pesos actuales es igual al peso máximo permitido.

2.2.2. Pseudocódigo

El caso base en este algoritmo se da cuando el índice i llega al final de la lista $items$ o cuando la variable w -que representa el peso que todavía se puede ocupar- llega a 0. La otra diferencia con el algoritmo de fuerza bruta es en la línea 4 de *solveBracktracking* donde se introduce la poda por factibilidad: si al restar el peso del ítem a w se obtiene un número negativo esto quiere decir que no se puede agregar el mismo sin excederse del límite, por lo que se omite su consideración para la solución final. Finalmente las llamadas recursivas en los demás casos son iguales al algoritmo de fuerza bruta.

Algorithm 3. Backtracking

```
1: procedure BACKTRACKING(int W, lista(tuplas) items)
2:     index ← 0
3:     solution ← 0
4:     return solveBacktracking(index, solution, W, items)
```

Algorithm 4. solveBacktracking

```
1: procedure SOLVEBACKTRACKING(int i, sol, w; lista(tuplas) &items)
2:     if (i = |items|) ∨ (w = 0) then
3:         return sol
4:     else if w − items[index].first < 0 then
5:         i ← i + 1
6:         return solveBacktracking(i,sol,w,items)
7:     else
8:         proximo ← i + 1
9:         conItem ← solveBacktracking(proximo, sol + items[i].second, w − items[i].first, items)
10:        sinItem ← solveBacktracking(proximo, sol, w, items)
11:        return max(conItem,sinItem)
```

2.2.3. Correctitud

El algoritmo de backtracking termina por el mismo argumento que el de Fuerza Bruta cuando i llega a $|items|$. Una diferencia con el algoritmo anterior es que si se cumple la guarda de la línea 4 se realiza una nueva llamada recursiva en la que no se explora una rama por no conducir a una solución viable. Previo a esta llamada i también se incrementa por lo que sigue valiendo la demostración anterior. Además se agrega otra condición para que el algoritmo termine y es cuando $w = 0$. Esta condición nos permite terminar antes en los casos que se haya alcanzado, pero no excedido, el límite de la mochila.

La demostración de correctitud es casi idéntica a la del algoritmo de Fueza Bruta salvando las siguientes diferencias: por un lado, las dos llamadas recursivas que consideran agregar o no un ítem se almacenan en variables y el algoritmo devuelve la mayor de estas dos; por otro lado, en vez de acumular el peso y beneficio de una combinación en un *par* almacena los beneficios en *sol*; por último, va restando los pesos a w, que empieza en el límite de la mochila y es decrementado cada vez que se agrega un ítem. Además, por la condición de la línea 4, no se consideran elecciones que pasen el límite para mejorar el desempeño del proceso. Al terminar, el algoritmo nos devuelve la solución de mayor beneficio de todas las combinaciones posibles que no sobrepasan el límite W.

2.2.4. Complejidad

La complejidad es $O(2^N)$. Se puede deducir de la misma manera que en el algoritmo anterior ya que en los casos donde no pueda realizar ninguna poda son prácticamente iguales y tienen el mismo comportamiento asintótico. Esto refuerza la idea del backtracking como una fuerza bruta "*inteligente*". Como veremos más adelante en la etapa de experimentación, si el backtracking puede podar muchas ramas su desempeño mejorará notablemente y se alejará de la cota del peor caso, de lo contrario se asemejará al de fuerza bruta.

2.3. Dynamic Programming

2.3.1. Descripción

La idea detrás de la programación dinámica es resolver un problema complejo reduciéndolo a una colección de subproblemas más simples, solucionando cada uno de estos una única vez y guardando estos resultados parciales. Cuando uno de estos subproblemas se repita, tendremos la solución correspondiente a mano y no habrá necesidad de volver a calcularla. Como el problema de la mochila exhibe una subestructura óptima, sabemos que resolver los subproblemas de manera óptima nos va a permitir encontrar una solución para el problema general. Para esto vamos a ir guardando los resultados parciales de una manera semejante a los algoritmos anteriores, pero haciendo *memoization* para ahorrar cálculos.

2.3.2. Pseudocódigo

Este algoritmo crea una matriz de $n(W+1)$ y la inicializa con -1 en todas sus celdas. Cada vez que querramos calcular la suma de pesos y beneficios de un subconjunto primero revisamos la matriz en la posición correspondiente. Si está marcada con un -1 realizamos el cálculo y lo guardamos en su posición para futuras referencias, si el valor ya fue calculado simplemente lo retornamos. Al final del algoritmo tendremos en cada combinación de $i = 1, 2, ..., n$ y $w = W + 1, W, ..., 1, 0$ la solución al problema de i ítems y mochila con límite de peso w. La solución al problema original estará en (n, W).

2.3.3. Correctitud

En cada recursión el algoritmo resuelve el subproblema considerando los primeros i elementos y una mochila de menor límite. Esto lo hace decrementando *index* en 1 con cada llamada y w con el peso del i-ésimo ítem. Al llegar cualquiera de los dos a 0 la recursión termina. Asimismo al término del algoritmo tenemos la solución al problema en la posición de la matriz ($index$,w), habiendo completado la matriz para todas las combinaciones de ítems y w posibles.

- Los casos base mencionados anteriormente son dos: cuando ya no hay ítems a considerar o cuando el peso límite de la mochila es 0. En consecuencia:
  - Si el índice es 0 y el peso del ítem actual es menor que w se actualiza la posición de la matriz $(0, w)$ con su beneficio, caso contrario se carga un 0 en la misma posición.
  - Si $w = 0$ se carga un 0 ya que el beneficio posible de una mochila con límite 0 es 0.
- Para los demás casos se evalúa si la mochila tiene espacio para el i-ésimo ítem:
  - Si no lo tiene, el beneficio está en la posición de la matriz (índice-1,w), que es la solución al subproblema con un elemento menos (los primeros $i - 1$) y mismo límite.
  - Si tiene espacio, la solución al problema es el máximo de calcular las soluciones de los subproblemas agregando o no el i-ésimo ítem. O sea, el máximo del subproblema que toma los primeros $i - 1$ elementos y tiene límite w menos el peso del i-ésimo, y el subproblema con los primeros $i-1$ elementos pero con el límite w sin modificar: $max(matriz[i-1][w-item.first]+item.second, matriz[i-1][w])$.

Vemos que se han agotado exhaustivamente todos los casos posibles y que calculamos el mayor beneficio de todas las combinaciones posibles de elementos, ya que cada vez que achicamos el problema consideramos el beneficio de agregar o no el ítem que estamos sacando.

2.3.4. Complejidad

El proceso de llenar la matriz sigue un diseño top-down en el que nunca se calcula el mismo subproblema más de una vez. En cada llamada recursiva se buscará la solución para la entrada de la matriz en la fila $n-1$ con y sin el ítem (lo que provoca el correspondiente movimiento de columna). Puesto que en la matriz tenemos todas las soluciones a los subproblemas, podemos mejorar sustancialmente los tiempos ya que no requieren ser computados nuevamente. Aunque su complejidad espacial sea mayor $O(nW)$, su complejidad temporal corresponde a la complejidad de completar la matriz $O(nW)$.

Algorithm 5. DP

```
1: procedure DP(int W, lista(tuplas) items)
2:     Matrix matrix[n][W + 1]                ▷ matrix es una matrix de nW inicializada con −1
3:     result ← solvDP(|items| − 1, W, matrix, items)
4:     return result
```

Algorithm 6. solveDP

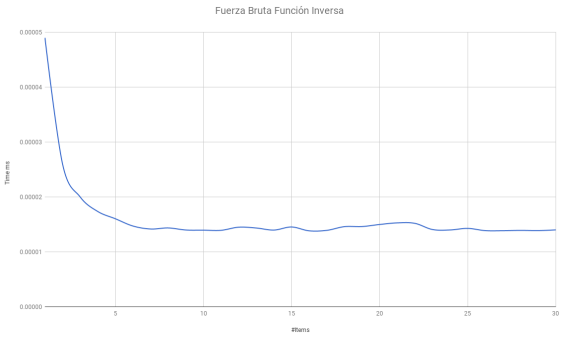
```
1: procedure SOLVEDP(int index, w, &matrix, &items)
2:     if maxtrix[index][w] ≠ −1 then
3:         return matrix[index][w]
4:     item ← items[index]
5:     if index = 0 then
6:         if item.first ≤ w then
7:             return item.second
8:         else
9:             return 0
10:    if w = 0 then
11:        return 0
12:    if item.first > w then
13:        return solveDP(index − 1, w, matrix, items)
14:    conItem ← solveDP(index − 1, w − item.first, matrix, items) + item.second)
15:    sinItem ← solveDP(index − 1, w, matrix, items)
16:    matrix[index][w] ← max(conItem, sinItem)
17:    return matrix[index][w]
```

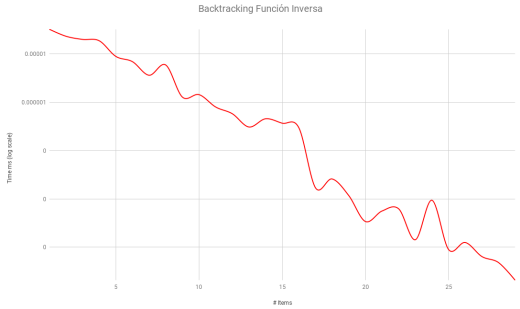
3. Experimentación

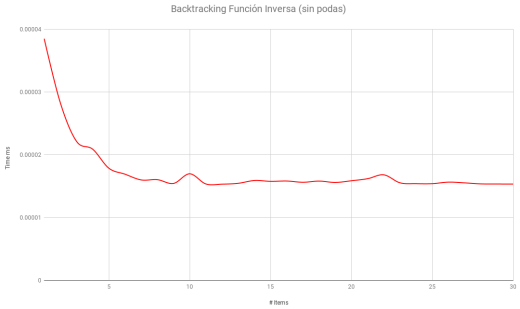
3.1. Complejidad temporal

Queremos comprobar las complejidades temporales de los tres algoritmos. Para hacerlo generamos aleatoriamente listas de distintas cantidades de elementos usando la función rand() de C++, evitando casos triviales con ítems de pesos positivos y menores al límite y beneficios mayores a 0. Cada vez que se corrió un algoritmo con una mochila se hizo 100 iteraciones y se las promedió. Observamos que mochilas de más de 30 ítems producían tiempos demasiado altos para el algoritmo de fuerza bruta, por lo que este algoritmo no se probó con entradas mayores a este número.

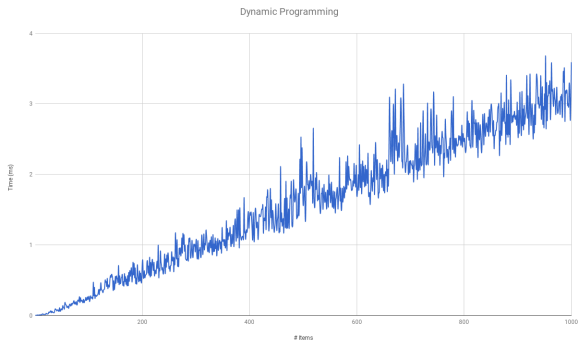
Buscamos respaldar empíricamente el cálculo de la complejidad para el algoritmo de Fuerza Bruta. Aplicando la función inversa $f(x) = \frac{x}{2^n}$ a los tiempos obtenidos y graficando estos resultados, logramos observar cómo la curva se aproxima a una constante, lo cual respalda nuestra hipótesis. Para el algoritmo de Backtracking vemos que el mismo procedimiento no lleva a los mismos resultados por las podas que tienen lugar (abajo a la izquierda). Al experimentando con valores de entrada que no producen podas, la aplicación de la función inversa sí nos devuelve una curva que se aproxima a una constante (abajo a la derecha).

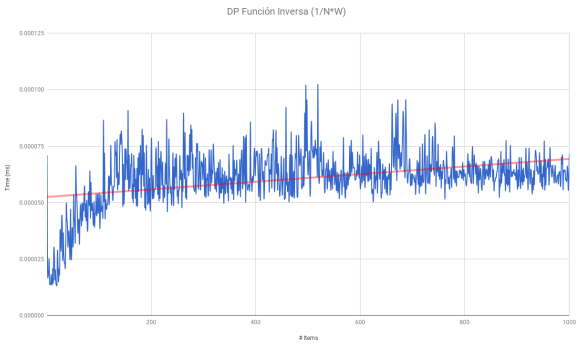


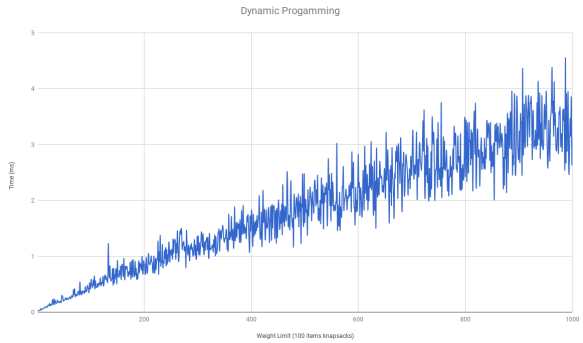


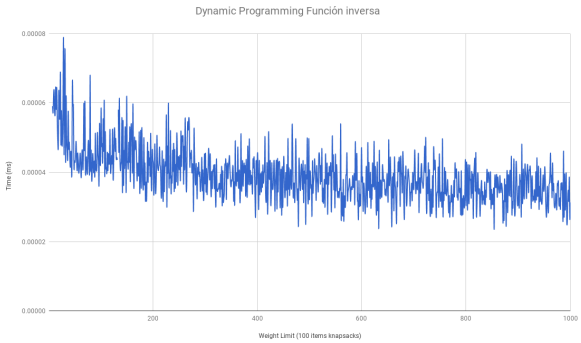


Para corroborar la complejidad calculada para el algoritmo de programación dinámica primero probamos dejando $W = 50$ fijo con mochilas de hasta 1000 elementos. En los dos gráficos siguientes podemos observar el crecimiento lineal de los tiempos de resolución y en la función inversa $f(x) = \frac{x}{nW}$ cómo se acerca a un valor constante a medida que crece n con un W fijo. Luego repetimos el experimento dejando fija la cantidad de elementos y variando W de 1 hasta 1000. Los resultados alcanzaron nuestras expectativas sobre la linealidad de su complejidad (cuando una variable permanece fija).

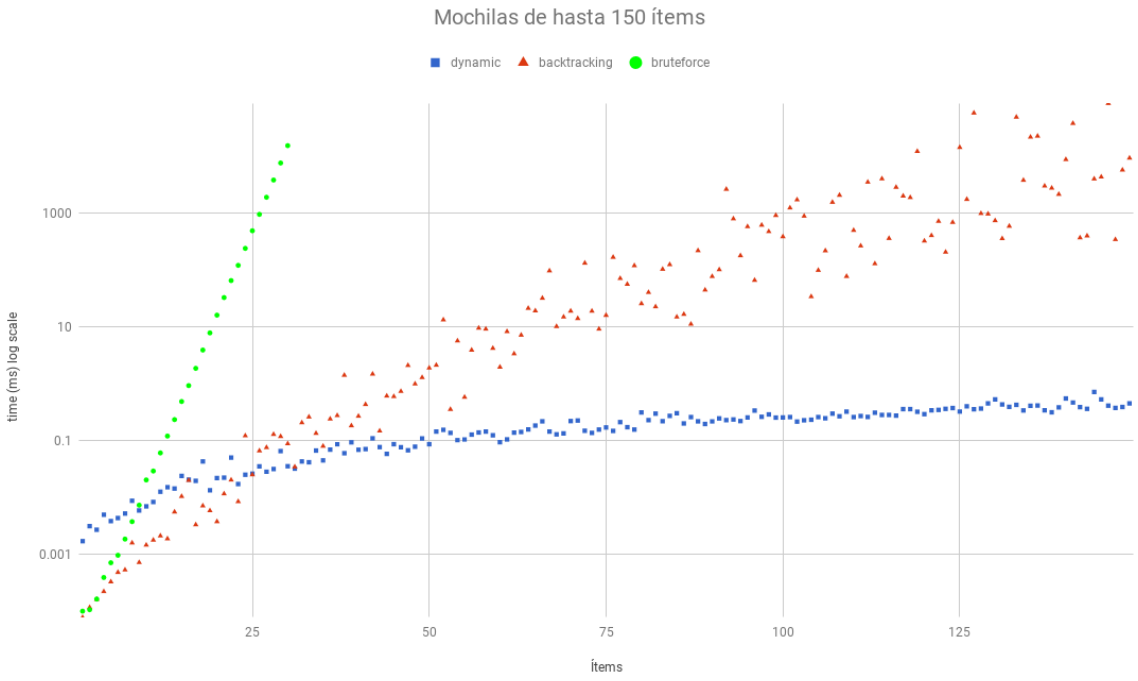








Finalmente probamos con listas de hasta 150 ítems (hasta 30 en el caso de la Fuerza Bruta), con pesos y beneficios entre 1 y 50 y W = 50. En estos casos, dada la distribución de pesos y el límite podemos comprobar la diferencia de comportamiento entre el Backtracking y la Fuerza Bruta debido a las podas por factibilidad que tienen lugar. Las observaciones son consistentes con la complejidad calculada para cada algoritmo y con los experimentos anteriores. Además, corroboramos que en casos promedio la complejidad del backtracking no es $O(2^n)$, si lo fuese tendríamos una recta como la del algoritmo de Fuerza Bruta -cabe aclarar que estamos usando una escala logarítmica que justamente es la función inversa de la exponencial-.



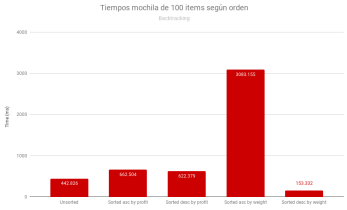
3.2. Efectos del sorting

Para encontrar mejores y peores casos buscamos analizar los tiempos de ejecución de los algoritmos frente a listas de ítems con diversos órdenes: por peso ascendente, por peso descendente, por beneficio ascendente, por beneficio descendente y ningún orden. En la página siguiente se encuentran los gráficos para los tres algoritmos en sus versiones de sortings para mayores cantidades de ítems.

3.2.1. Sorting para Brute Force

Se esperaba que el orden de la lista previo a la búsqueda de la solución por medio del algoritmo de fuerza bruta no modificase los tiempos significativamente. La primera prueba consistió en aplicar los ordenamientos con una mochila de 25 ítems; como resultado, las variaciones en los tiempos registrados fueron menores al 1 %. En la prueba siguiente utilizamos 100 mochilas de 25 elementos y medimos los tiempos con las diferentes opciones de sortings. En el 95 % de los casos, ordenar las listas por peso ascendente registró peores tiempos que no aplicar orden alguno, sin embargo la diferencia en tiempos es próxima al 1 %, lo cual es despreciable si tenemos en cuenta el tiempo previo necesario para ordenar las listas. En el resto de los sortings, no podemos apreciar ninguna tendencia clara que contradiga nuestras expectativas por medio de los datos recolectados.

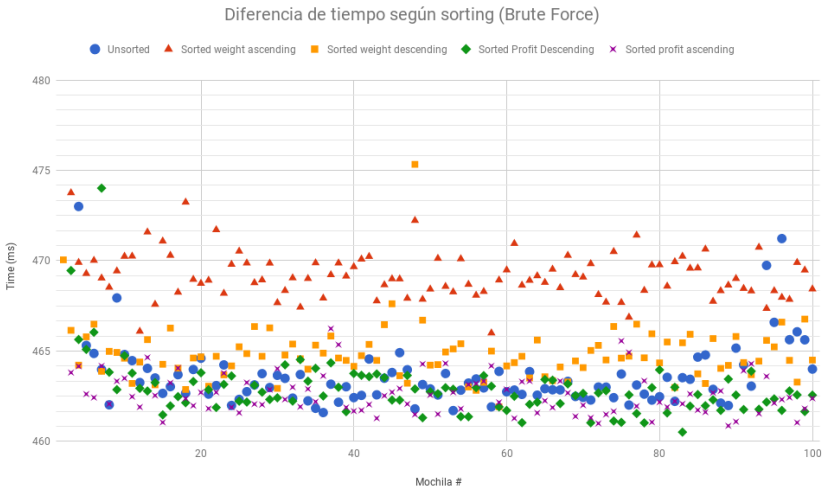
3.2.2. Sorting para Backtracking

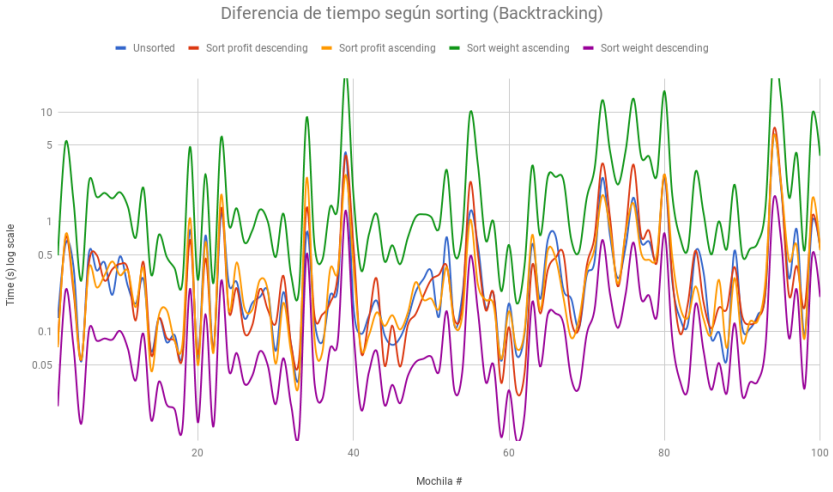


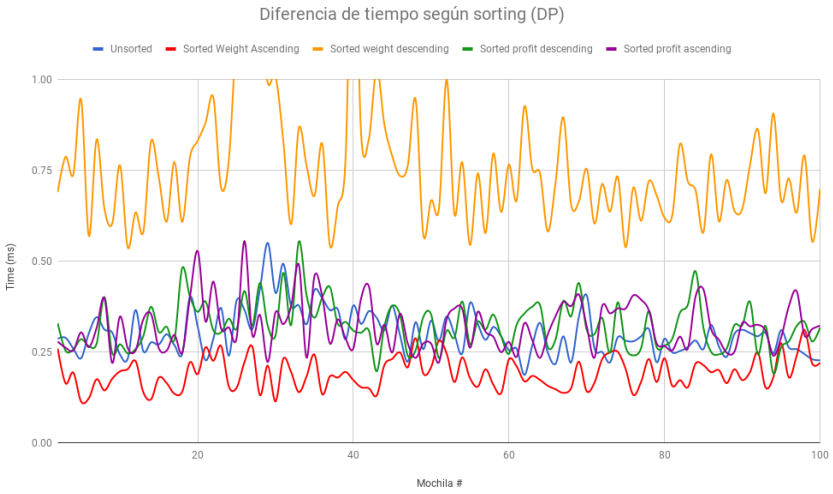
A través de la prueba con una sola mochila observamos que ordenarla por peso ascendentemente empeoró el tiempo de resolución del algoritmo, mientras que el ordenamiento inverso mejoró el tiempo del mismo. Llevando la misma prueba a 100 listas de 100 elementos cada una, vemos que la tendencia se mantiene. En todos los casos registrados ordenar las mochilas ascendentemente empeoró los tiempos de ejecución y hacerlo descendentemente los mejoró. El orden por beneficio no muestra una tendencia definida.

3.2.3. Sorting para DP

La primera prueba con una mochila da indicios de cómo el orden por peso puede influir en los tiempos de ejecución del algoritmo. Se pudo observar en el 100 % de los casos que ordenar los elementos por peso descendente empeora los tiempos, mientras que ordenarlos ascendentemente introdujo leves mejoras en el 95 % de las observaciones. Como era esperable no se detectan tendencias definidas al ordenar por beneficio.


Diferencia de tiempo según sorting (Brute Force)
Unsorted
Sorted weight ascending
Sorted weight descending
Sorted Profit Descending
Sorted profit ascending
480
475
470
465
460
Time (ms)
20
40
60
80
100
Mochila #


Diferencia de tiempo según sorting (Backtracking)
Unsorted
Sort profit descending
Sort profit ascending
Sort weight ascending
Sort weight descending
10
5
1
0.5
0.1
0.05
Time (s) log scale
20
40
60
80
100
Mochila #


Diferencia de tiempo según sorting (DP)
Unsorted
Sorted Weight Ascending
Sorted weight descending
Sorted profit descending
Sorted profit ascending
1.00
0.75
0.50
0.25
0.00
Time (ms)
20
40
60
80
100
Mochila #

4. Conclusiones

Pudimos observar con las complejidades calculadas y los experimentos realizados que los algoritmos de fuerza bruta -y en algunos casos los de backtracking- son inviables para resolver problemas combinatorios con valores de entrada grandes. También podemos apreciar que, en problemas de procesos de toma de decisiones, podar ramas que no llevan a una solución válida es una buena estrategia para mejorar el rendimiento de un algoritmo. Esta simple diferencia explorada en el backtracking muestra que, en caso de que los datos posibiliten la existencia de podas, su rendimiento es mucho mejor, manteniéndose competitivo incluso con la implementación de programación dinámica para problemas de n chico. Sin embargo, para poder aprovechar mejor los efectos de las podas, deberíamos saber de antemano algo sobre la distribución de los valores de entrada, o de modo contrario nos arriesgaríamos a que performe como una fuerza bruta.

También comprobamos que la técnica de programación dinámica tiene una mejor complejidad temporal, haciendo viables problemas que son imposibles bajo un acercamiento por fueza bruta (con o sin podas). Si bien lo hace con mayores requerimientos espaciales, una variante bottom-up podría bajar dicho requerimiento, lo cual resultaría muy beneficioso si el valor de nW fuese muy grande.

Finalmente pudimos ver el efecto que tiene ordenar los elementos antes de intentar buscar una solución para un W determinado. Habría que considerar la opción de ordenar por peso ascendente si tenemos grandes volúmenes de datos con los que trabajar y a partir de los cuales se querrá calcular múltiples mochilas.

En conclusión, los tres algoritmos presentados en este trabajo encuentran la solución óptima a las instancias del knapsack problem, por lo que son alternativas de enfoque correctas aunque con complejidades diferentes. La viabilidad de cada uno dependerá de nuestras necesidades en términos de complejidad espacial y temporal, de nuestro conocimiento previo sobre los datos con los que trabajaremos, y de la cantidad de soluciones a distintas mochilas con peso W que busquemos sobre una misma lista de elementos.