Dame la mochila!

Informe TP1

Joaquin Romera LU: 183/16
joakromera@gmail.com

April 2018

1 Introduction

1.1 Introduccion al 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:

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

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

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.

1.2 Introduccion a los algoritmos

A continuación consideraremos tres algoritmos que permiten encontrar soluciones a distintos knapsack problems: un algoritmo por fuerza bruta donde exploramos todas las combinaciones de items posibles (sin importar si una combinación particular está dentro de las restriccioneso no), una versión de backtracking donde calculamos los beneficios de todos los subconjuntos que respeten el límite de la mochila, y finalmente una implementación según la técnica de programación dinámica en la cual guardamos resultados parciales para ahorrar tiempo de computo en cálculos subsiguientes.

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 items y eligiendo la de beneficio más grande 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 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 este 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 index en cada iteración y llamandose 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.

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)        ▷ Agregamos item
9:         par.first ← items[index − 1].first
10:        par.second ← items[index − 1].second
11:        partialSums(index, par, solution, w, items)        ▷ No agregamos item
```

2.1.3 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, deteniendo el cálculo de una rama cuando una solución parcial no puede progresar. Esta "poda" por factibilidad se da cuando la suma parcial actual se excedería 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 es 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 linea 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 recursivos 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(index,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 Complejidad

La complejidad es $O(2^N)$. Se puede deducir de la misma manera que en el algoritmo anterior ya que para casos donde no pueda realizar ninguna poda tienen el mismo comportamiento asintótico.

2.3 Dynamic Programming

2.3.1 Descripción

Vamos a ir guardando lo resultados parciales de una manera semejante a los algoritmos anteriores, pero haciendo memoization para ahorrar cuentas. Para eso, este algoritmo crea una matriz de $n(W+1)$ y la inicializa con -1 en todas sus celdas.

2.3.2 Pseudocódigo

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.

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:        return solveDP(index, sol, w, items)
15:    conItem ← solveDP(index − 1, w − item.first, matrix, items) + item.second)
16:    sinItem ← solveDP(index − 1, w, matrix, items)
17:    matrix[index][w] ← max(conItem, sinItem)
18:    return matrix[index][w]
```

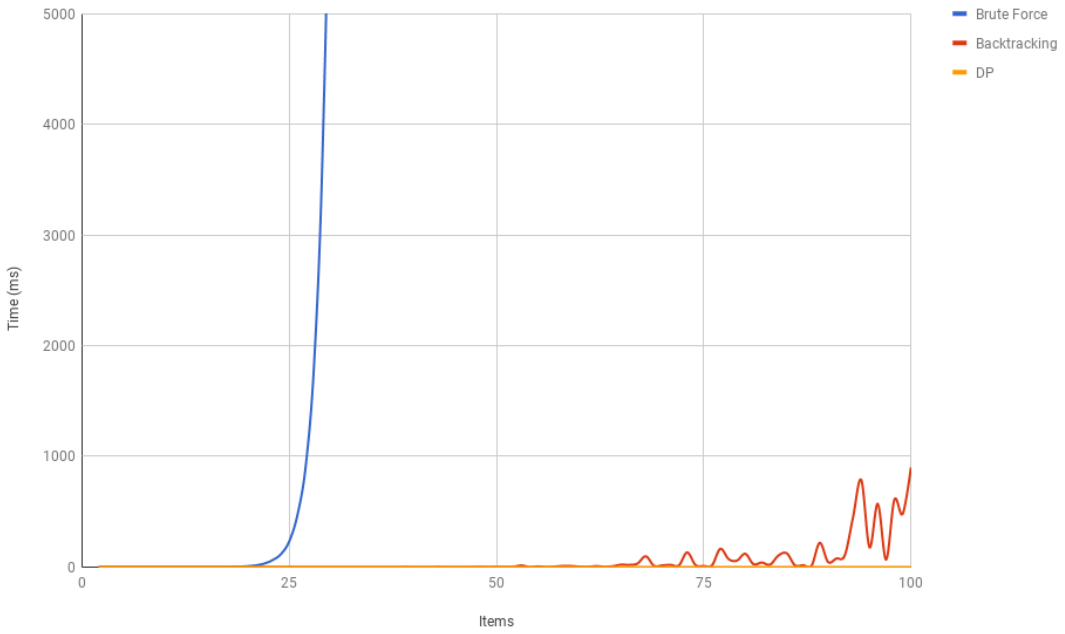
2.3.3 Complejidad

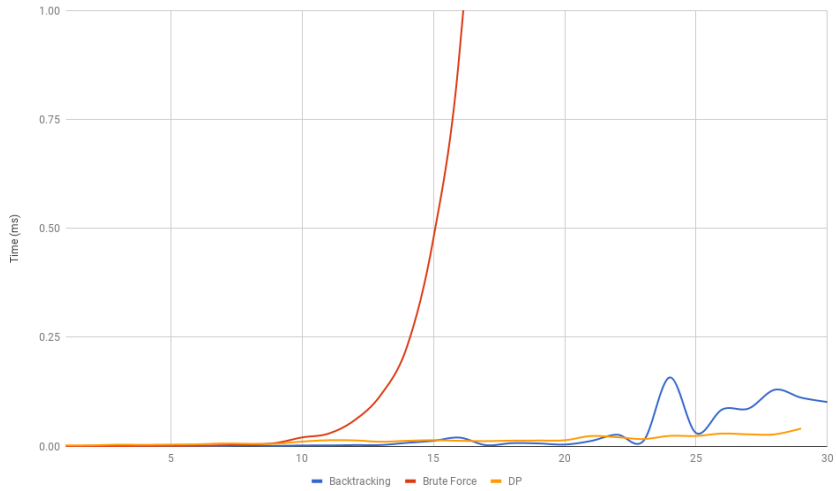
El proceso de llenar la matriz sigue un diseño top-down en el que nunca se calcula el mismo subproblema más de una vez. Esto permite mejorar sustancialmente los tiempos aunque su complejidad espacial sea mayor ($O(nW)$), la complejidad temporal del mismo es la complejidad de completar la mariz ($O(nW)$).

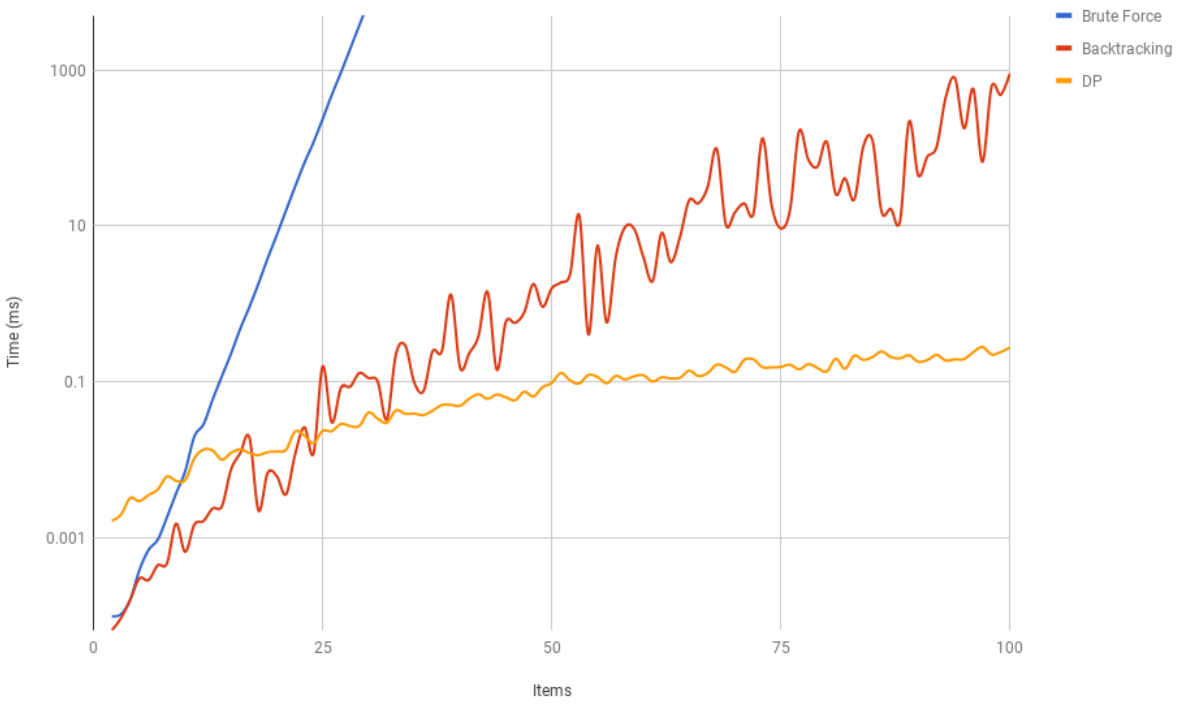
3 Experimentación

3.1 Complejidad temporal

Queremos comprobar las complejidades temporales de los tres algoritmos. Para hacerlo generamos aleatoriamente mochilas de hasta 100 elementos usando la función rand() de C++. Cada vez que se corrió un algoritmo con una mochila se hizo 100 iteraciones y se las promedió.
Se puede apreciar que los crecimientos son consistente con la complejidad calculada. La tercer imagen corresponde a una escala logarítmica.





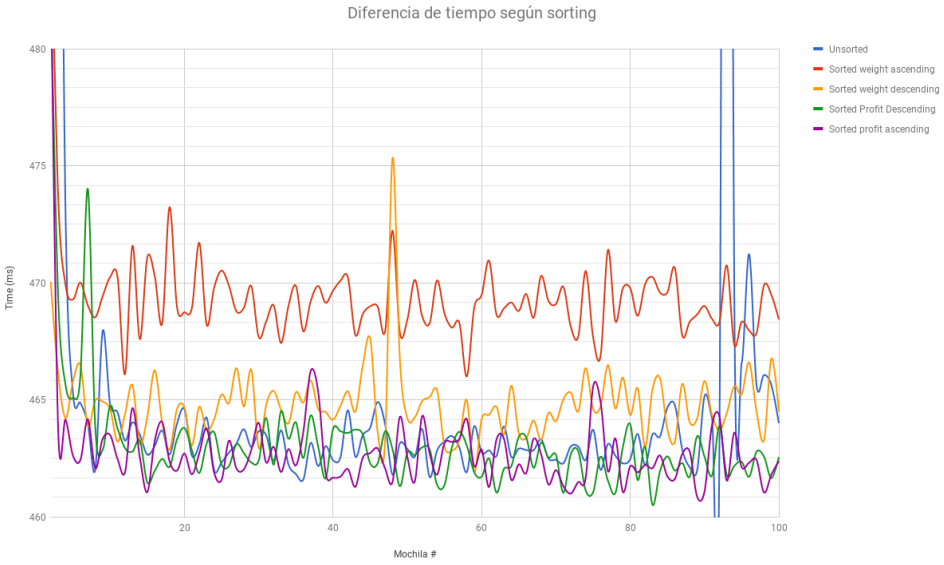


3.2 Sorting

Para encontrar mejores y peores casos buscamos analizar los tiempos de ejecución de los algoritmos frente a listas de items con diversos ordenes: ordenada por peso ascendente, por peso decendente, por beneficio ascendente, beneficio descendente y ningún orden.

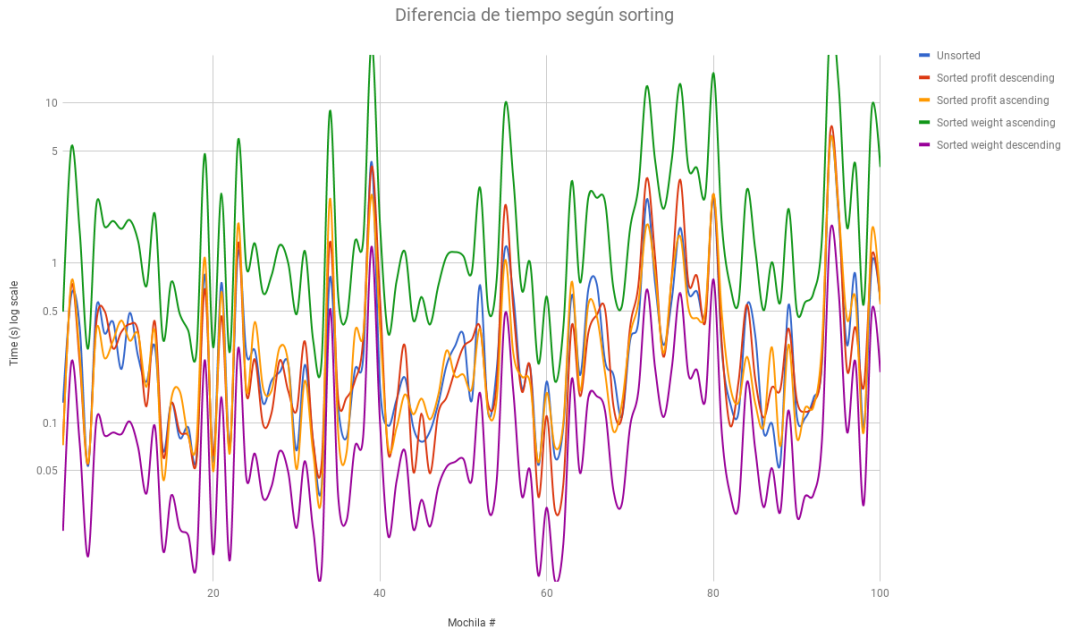
3.2.1 Sorting para Brute Force

Esperaríamos que ordenar la lista antes de buscar la solución por medio del algoritmo de fuerza bruta no mejorase los tiempos.



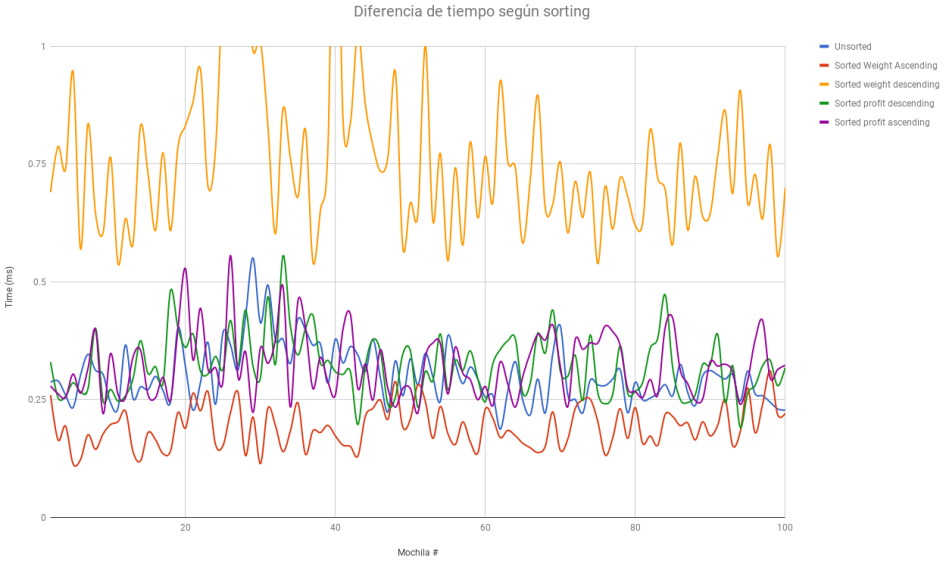
Efectivamente no podemos apreciar ninguna tendencia por medio de los datos recolectados.

3.2.2 Sorting para Backtracking



Parecería que los tiempos de ejecución empeoran y mejoran consistentemente según se ordene por peso ascendentemente o descendentemente. En el gráfico de abajo vemos como cambian lo tiempso para una mochila de 100 elementos.

3.2.3 Sorting para DP



Se puede ver que ordenarlos por peso descendente empeora los tiempos notablemente, mientras que ordenarlos ascendentemente introduce leves mejoras. Como era esperable no hay cambios al ordenar por beneficio.

4 Conclusiones

Por un lado podemos apreciar que en problemas de procesos de tomas de decisiones podar ramas que no llevan a una solución no contribuye es una buena estrategia para poder resolver problemas de alta complejidad en casos promedio.
Al mismo tiempo guardar resultados que sabemos que podemos llegar a necesitar más adelante en otros cálculos es otra manera de mejorar la eficiencia de un algoritmo.
La combinación de ambas estrategias permite encontrar soluciones para problemas que serían muy inviables por algoritmos de fuerza bruta.