Memoria Práctica 4 - Problema de la mochila simple

Programación dinámica

Por:
Alejandro Madueño Buciegas
León Elliott Fuller
Antoni Martí Albons

En esta práctica construimos una solución al problema de la mochila simple mediante programación dinámica. Esa construcción se estructura de la siguiente manera:

1. ¿Cumple los requisitos para poderse resolver con programación dinámica?
2. Ecuación recurrente del problema
3. Almacenamiento de soluciones parciales
4. Principio de optimalidad de Bellman
5. Diseño del algoritmo
6. Implementación del algoritmo
7. Cálculo de la eficiencia
8. Ejemplo

1. ¿Cumple los requisitos para poderse resolver con programación dinámica?

Las dos propiedades principales que se han de ser aplicables a un problema para poder abordarlo con programación dinámica son:

- Subproblemas superpuestos: se ha de poder descomponer en subproblemas que se repitan.
- Subestructura óptima: se ha de poder encontrar la solución óptima para el problema usando las soluciones (también óptimas) para sus subproblemas.

¿Cumple el problema de la mochila con estos requisitos? Podemos emplazar con una mochila pequeña y con solo un objeto, e ir incrementando el tamaño de la mochila y el número de objetos, preguntándonos cada vez que hacemos esto, si mejoramos las soluciones anteriores añadiendo el objeto nuevo, o si el nuevo tamaño de la mochila nos permite almacenar un objeto que previamente no podía; o de lo contrario, si sacando objetos para meter el nuevo empeora la solución que ya teníamos.

Este enfoque si cumple con el principio de los problemas superpuestos, pues para calcular la solución óptima para cada subproblema necesitamos de algunas de las soluciones óptimas para sus subproblemas (que a su vez utilizan soluciones a sus subproblemas), por tanto, es posible necesitaremos consultar el valor de las soluciones para los subproblemas varias veces. Vemos así que al dividir el problema en subproblemas, aparece solapamiento entre estos.

La justificación de que se cumple con la subestructura óptima se ve en el punto 4.

2. Ecuación recurrente del problema

Tratamos de formalizar el problema mediante una ecuación que relacione el problema actual con los anteriores.

Nomenclatura:

- n: número de objetos del problema.
- w_i: peso del objeto i.
- v_i: valor del objeto i.
- W: Peso máximo que puede cargar la mochila.
- m: $mcd(w_1, w_2, \dots, w_n)$.

Sea S una función que toma como entrada una pareja (a, p), donde a es un entreo menor que n. Llamamos p' a $\lfloor p/m \rfloor \cdot m$, es el máximo peso usable en el problema. En esta función, a nos indica de cuantos objetos disponemos para formar la solución (los a primeros) y p el peso máximo que puede cargar la mochila para la que estamos buscando la solución, lo que devuelve S es la solución óptima al problema con esas restricciones.

Podemos definir a S de la siguiente manera:

$$S(a, p) = \begin{cases} -\infty & \text{si } p < 0 \\ 0 & \text{si } p \geq 0 \wedge a = 0 \\ S(a, p') & \text{si no, si } p \neq p' \\ \max(S(a-1, p), S(a-1, p-w_a) + v_a) & \text{en cualquier otro caso} \end{cases}$$

La solución venimos buscando viene dada por $S(n, W)$.

3. Almacenamiento de soluciones parciales

Al plantearnos de qué manera se pueden almacenar las soluciones parciales, no se tarde mucho en ver que una tabla, o matriz bidimensional, es una buena opción (más allá de ser la que se nos pide que implementemos). Esto es porque trabajamos con una función que toma como entrada dos valores, y que agrupa los valores que toma para estos (no es continua). Se propone la siguiente tabla, donde cada casilla es un valor de S:

	$0 \cdot m$	$1 \cdot m$	$2 \cdot m$	…	p'
1	$S(1,0 \cdot m)$	$S(1,1 \cdot m)$	$S(1,2 \cdot m)$	…	$S(1,p')$
2	$S(2,0 \cdot m)$	$S(2,1 \cdot m)$	$S(2,2 \cdot m)$	…	$S(2,p')$
…	…	…	…	…	…
n	$S(n,0 \cdot m)$	$S(n,1 \cdot m)$	$S(n,2 \cdot m)$	…	$S(n, p')$

4. Principio de optimalidad de Bellman

El principio de optimlidad de Bellman establece que:

> *"Una política óptima tiene la propiedad de que cualquiera que sea el estado inicial y la decisión inicial, las decisiones restantes deben constituir una política óptima en relación con el estado resultante de la primera decisión."*

Esto es fácil de ver en nuestro problema. Teniendo la solución para n objetos, si queremos añadir otro, solo lo usamos si mejora la solución que ya teníamos, obteniendo así la solución óptima para $n+1$ objetos. Exactamente igual pasa aumentando el peso. Por tanto, sigue una subestructura óptima y cumple con el principio de optimalidad.

5. Diseño del algoritmo

Nótese que para calcular el valor una casilla de posición (f, c) necesitamos los valores de las posiciones $(f\text{-}1, c)$ y $(f\text{-}1, c')$, siendo $c' = c - w_f$. Buscamos $S(n, W)$ y

no es difícil ver que $S(n, W) = S(n, p')$. Si $m < \min(w_1, w_2, \dots, w_n)$, o lo que es lo mismo, si $m \neq \min(w_1, w_2, \dots, w_n)$, la columna anterior a la que pertenece a $S(n, p')$ no se sua para generar $S(n, p')$, ya que al cambiar a las columnas p' - w_i nunca iremos a la p' - m, porque todos los w son mayores que m. Por eso no conviene llenar toda la tabla y después calcular la solución, sino ir calculando solo las casillas necesarias. Al calcularlas almacenamos su valor en la tabla, para que si necesitamos acceder a ella de nuevo no tengamos que volver a calcularla.

Pseudocódigo para la solución podría ser:

```
global tabla

ValorMaximo(peso_max, lista_items) {
        m = mcd(lista_items.pesos)
        peso = floor(peso_max / m) * m

        return ValorMaximo(lista_items.size, peso, lista_items)
}

ValorMaxino(num_items, peso_max, lista_litems) {
        si peso_max < 0   return -∞
        si num_items = 0  return 0

        si existe tabla[num_items][peso_max] {
                return tabla[num_items][peso_max]
        }

        item = lista_items[num_items -1]
        valor_item = item.valor
        peso_disp_con_item = peso_max - item.peso
        sin_obj = ValorMaxino(num_items-1, peso_max, lista_items)
        con_obj = ValorMaxino(num_items-1, peso_disp_con_item, lista_items) + valor_item
        tabla[num_items][peso_max] = max(sin_obj, con_obj)

        return tabla[num_items][peso_max]
}
```

Las entradas de columna de la tabla pueden substituirse por el número de columnas si se accede al peso deseado dividiendo el índice por m. Aunque esto no es necesario, puede ahorrar espacio dependiendo de como se implemente la tabla.

6. Implementación del algoritmo

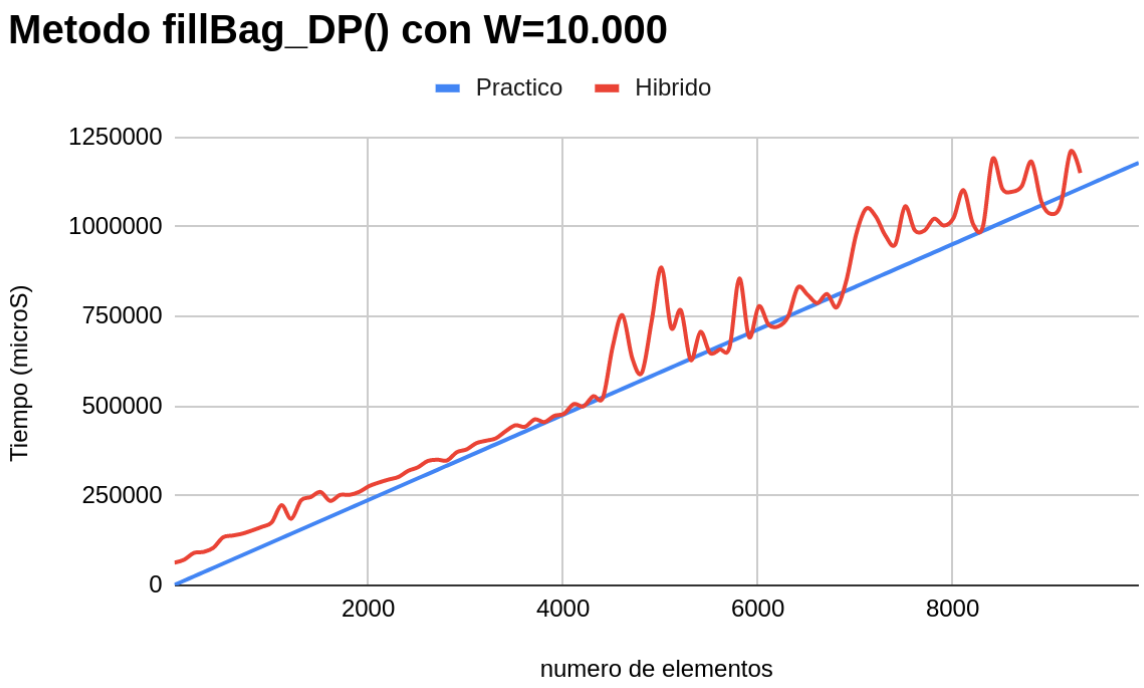
En c++. A poder ser, se ha de proporcionar alguna manera de cargar el problema desde un fichero de texto; si es editable y legible **FÁCILMENTE** por las personas, mejor.

7. Cálculo de la eficiencia

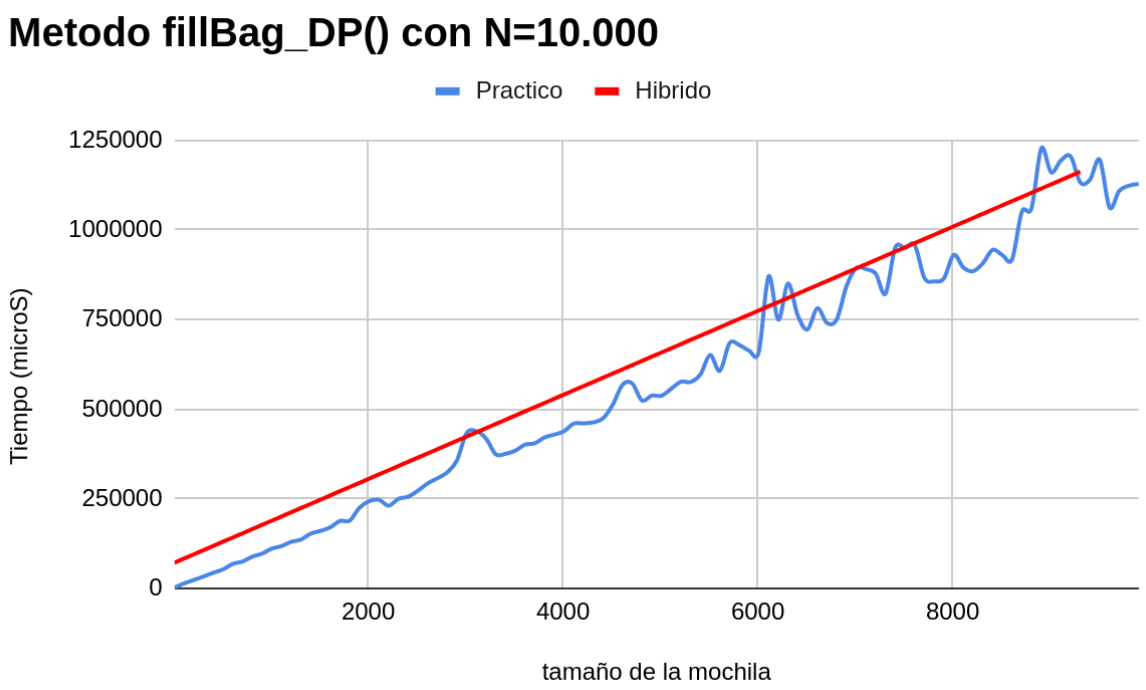
El algoritmo tiene una eficiencia teórica de Θ(n.w), ya que, la tabla tiene (n+1)(w+1) entradas, siendo w el valor de la capacidad de la mochila y n el número de objetos a tener en cuenta, con cada entrada teniendo un cálculo de cómputo de Θ(1).

Como la eficiencia es cuadrática, hemos pensado que la mejor manera de demostrar la eficiencia práctica es haciendo uso de dos gráficas distintas, donde en una se varía el número de ítems dejando el peso de la mochila constante, y de la misma manera, la otra gráfica muestra la variación del peso de la mochila frente a un número constante de ítems.

Gráfica con un número w constante, tamaño de mochila constante.



Gráfica con un número n constante, número de elementos constante.



Podemos observar en ambas gráficas, que se comportan como funciones lineales, pudiendo confirmar nuestra hipótesis de que el algoritmo fillBag_DP() tiene una complejidad cuadrática Θ(n.w).
Los datos obtenidos son mediante la ejecución de **make DBP-W y make DBP-N**

8. Ejemplo

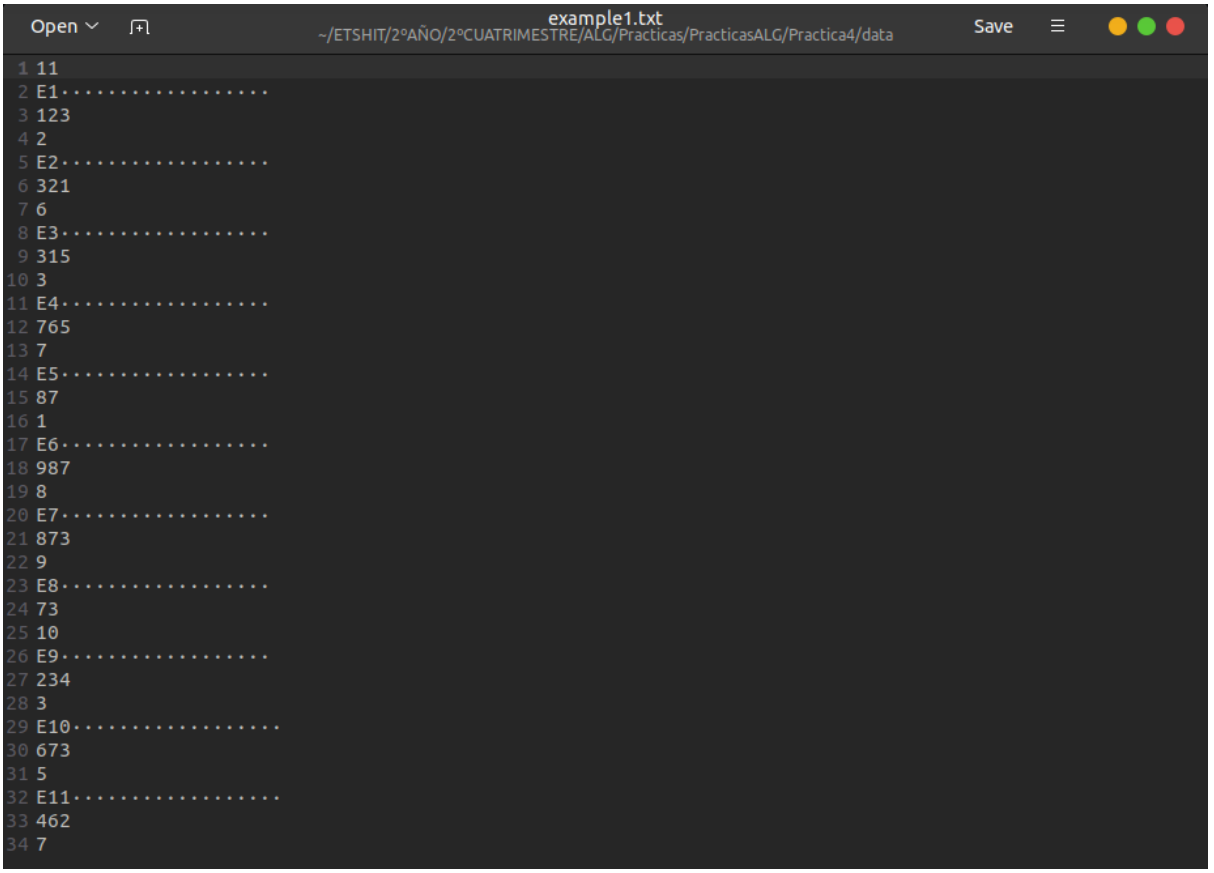
Se muestran dos ejemplos para instancias del problema que se pueden cargar desde un fichero de texto.

A través del makefile se pueden ejecutar los dos ejemplos presentados a continuación, que se ejecutan a través de un fichero main denominado DealerBagProblem.cpp localizado en la carpeta /src.

Argumentos de ejecución:

Use f <FILE> | b (10000 elements | 1000000 columns | 1 column step) | d < 1 (STIMULANTS) | 2 (OPIOIDS) | 3 (DEPRESSANTS) | 4 (HALLUCINOGENS) | 5 (DISSOCIATIVES) | 6 (INHALANTS) | 7 (CANNABIS) > && <v (INFO) | n (NO INFO)>"

Primer ejemplo:



Este fichero muestra una instancia de un ejemplo siguiendo un formato de entrada:

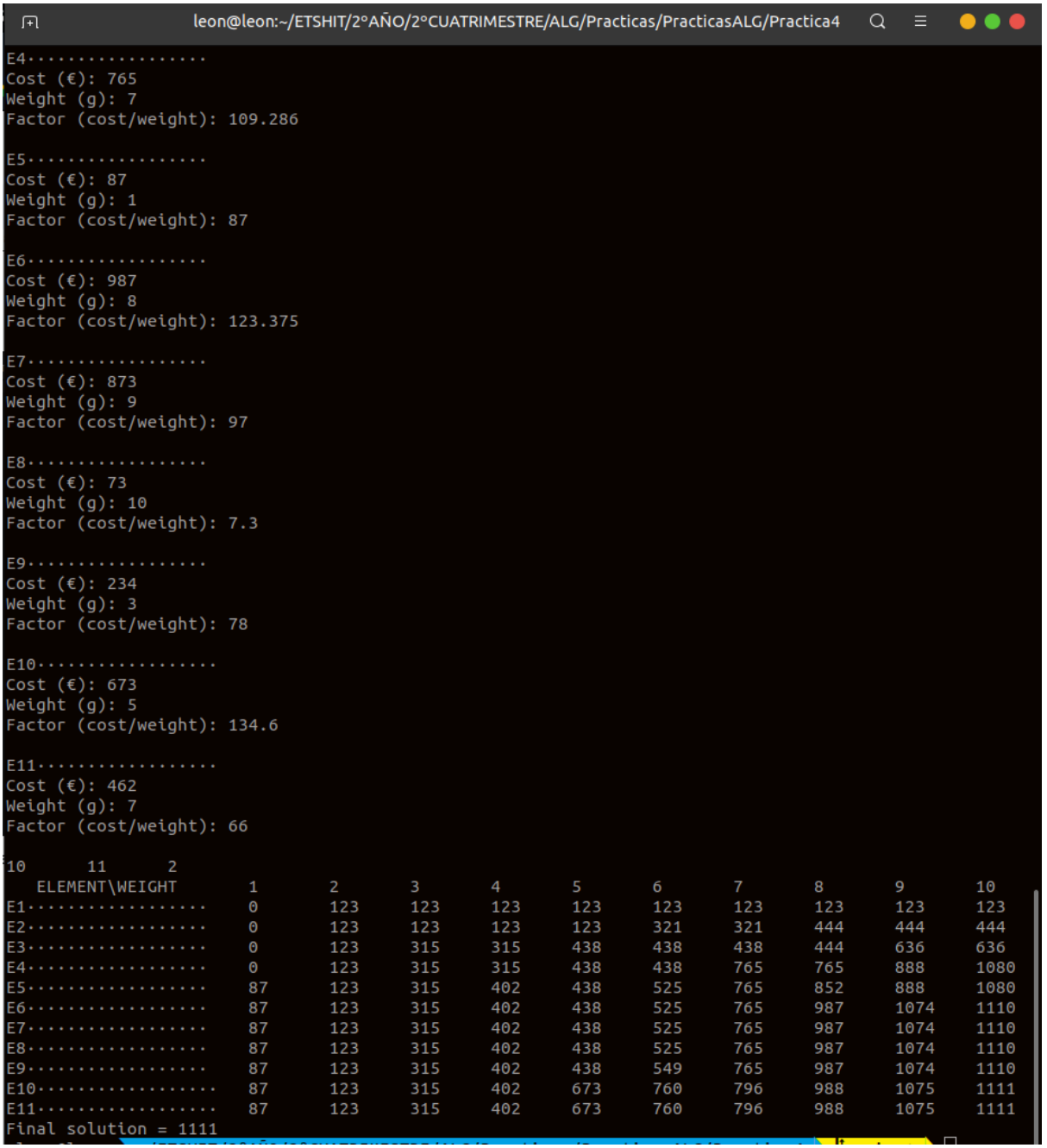
1.Número de ítems

2.Nombre primer ítem

3.Precio primer ítem

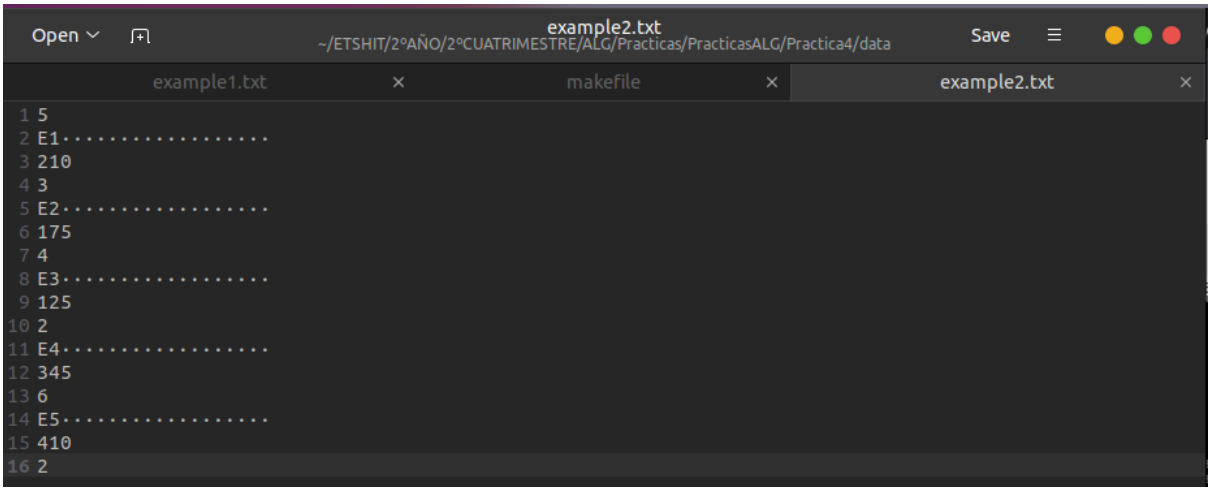
4.Cantidad primer ítem

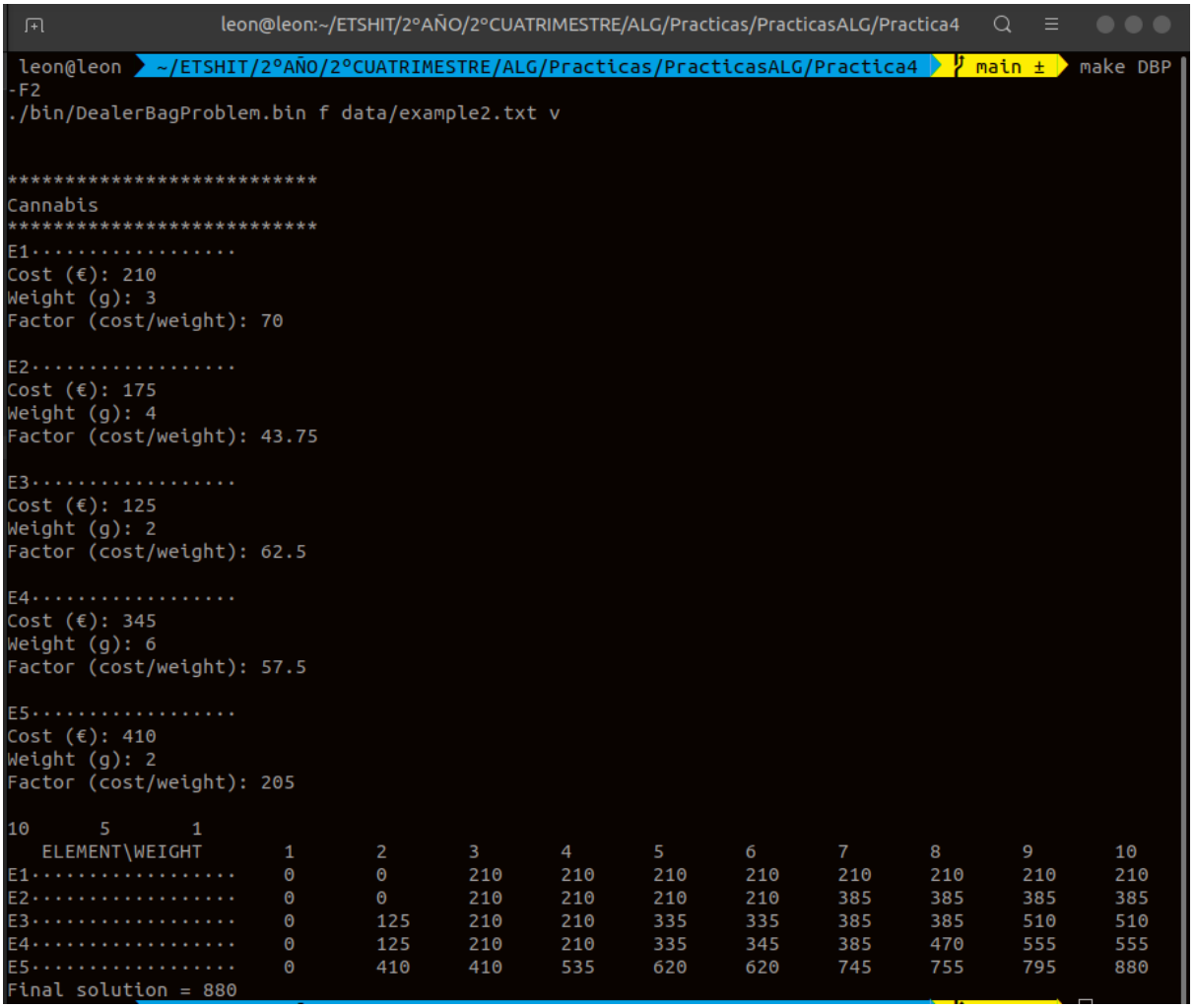
Ejecutando **make DBP-F1**:



Segundo ejemplo:

Al igual que antes, con un fichero de entrada para cargar los datos, y ejecutando el comando **make DBP-F2**:


Open
example2.txt
~/ETSHIT/2ºAÑO/2ºCUATRIMESTRE/ALG/Practicas/PracticasALG/Practica4/data
Save
example1.txt
makefile
example2.txt
5
E1··················
210
3
E2··················
175
4
E3··················
125
2
E4··················
345
6
E5··················
410
2


leon@leon:~/ETSHIT/2ºAÑO/2ºCUATRIMESTRE/ALG/Practicas/PracticasALG/Practica4
leon@leon ~/ETSHIT/2ºAÑO/2ºCUATRIMESTRE/ALG/Practicas/PracticasALG/Practica4 main ± make DBP
-F2
./bin/DealerBagProblem.bin f data/example2.txt v
****
Cannabis
****
E1··················
Cost (€): 210
Weight (g): 3
Factor (cost/weight): 70
E2··················
Cost (€): 175
Weight (g): 4
Factor (cost/weight): 43.75
E3··················
Cost (€): 125
Weight (g): 2
Factor (cost/weight): 62.5
E4··················
Cost (€): 345
Weight (g): 6
Factor (cost/weight): 57.5
E5··················
Cost (€): 410
Weight (g): 2
Factor (cost/weight): 205
10 5 1
ELEMENT\WEIGHT 1 2 3 4 5 6 7 8 9 10
E1·· 0 0 210 210 210 210 210 210 210 210
E2·· 0 0 210 210 210 210 385 385 385 385
E3·· 0 125 210 210 335 335 385 385 510 510
E4·· 0 125 210 210 335 345 385 470 555 555
E5·· 0 410 410 535 620 620 745 755 795 880
Final solution = 880