



UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

MÉTODOS ALGORÍTMICOS EN RESOLUCIÓN DE PROBLEMAS

Comparación de algoritmos. Problema de la mochila.

Autor:

Jorge Villarrubia Elvira

Segundo cuatrimestre

14 de abril de 2019

Índice

1. Introducción	2
2. Algoritmo devorador	3
2.1. Pruebas con elementos aleatorios	3
2.2. Prueba sesgada favorable	4
3. Algoritmo de programación dinámica	6
3.1. Prueba con tamaño de objetos y mochila variable	6
3.2. Tamaño de mochila fijo	7
3.3. Análisis de la pila de llamadas	8
4. Algoritmo de fuerza bruta	10
4.1. Prueba aleatorios	10
4.2. Análisis de memoria	11
5. Algoritmo de vuelta atrás	13
5.1. Poda por factibilidad	13
5.2. Poda por optimalidad	15
5.3. Combinación de podas y conclusiones	17
6. Algoritmo de ramificación y poda	18
6.1. Poda Optimista-pesimista con el voraz	18
6.2. Coste en memoria	20
7. Comparativas finales	21

1. Introducción

En lo sucesivo se hará un análisis lo más preciso y exhaustivo posible de los distintos algoritmos que resuelven el famoso problema de la mochila.

A modo de introducción se adjunta a continuación el enunciado más general del mismo.

Problema de la mochila

Se tiene una mochila que es capaz de soportar un peso máximo P , así como un conjunto de objetos, cada uno de ellos con un peso y un beneficio.

Se pretende obtener el máximo beneficio, guardando los objetos adecuados en la mochila.

Cuya versión puramente matemática es la siguiente:

Formulación matemática del problema

Sujeto a:

$$\begin{aligned} & \text{Máx} \sum_{i=1}^n x_i v_i \\ & \sum_{i=1}^n x_i p_i \leq P \end{aligned}$$

Donde x_i representa el objeto i -ésimo, y p_i , v_i , su peso y valor, respectivamente.

Nótese que se trata de un enunciado suficientemente ambiguo como para recibir distintas interpretaciones y de un problema suficientemente ‘jugoso’ como para que merezca la pena analizarlo desde diversos algoritmos. Para ello se tratarán, en general, casos grandes y suficientemente aleatorios que permitan obtener conclusiones interesantes.

Cada algoritmo será analizado empíricamente por separado, señalándose sus pros y contras. Finalmente, se hará una comparación simultánea de todos ellos con el fin de determinar cual es mejor para cada caso e intentando comprender el porqué.

2. Algoritmo devorador

Como ya se mencionó en la introducción, el enunciado del problema es suficientemente laxo como para que pueda interpretarse de varias formas. En caso de considerarse los objetos como fraccionables existe un algoritmo voraz bastante sencillo y con un coste tanto en tiempo como en espacio muy interesante que vamos a poner a prueba.

La implementación del mismo, de la que vamos a extraer las conclusiones, se puede encontrar en [4]. Para ser mas eficientes, la función de resolución machaca el vector de objetos ordenando sobre él mismo para evitar la copia. En principio, esta solución tiene coste $O(n * \log(n))$ en tiempo y lineal en memoria respecto al numero de datos. Muy bueno, aunque asumiendo quizás demasiado con la fraccionabilidad de objetos.

2.1. Pruebas con elementos aleatorios

Para las primeras pruebas vamos a considerar una cantidad de objetos suficientemente grandes de pesos y valores aleatorios, dentro de uno rangos razonables. El tamaño de la mochila también se ajustará un poco por debajo de lo que sería el promedio de peso por el número de objetos para que no quepan todos pero si suficientes como para que le problema se ajuste a un caso real. Los valores escogido son:

$$N = 300.000 \quad 1.0 \leq p_i \leq 5.0 \quad \text{Promedio } p_i = \frac{5.0 + 1.0}{2} = 3.0$$

$$M = (\text{Promedio } p_i - 1.0) * N = 2 * (300.000) = 600.000$$

Tras ejecutarse varias veces, se aprecia que se meten en la mochila unos 220.000 objetos (enteros o fraccionados), con lo que la prueba parece realista y bien definida. Era nuestra intención meter muchos objetos, pero no todos. Mientras que el tiempo de ejecución es de apenas segundo y medio. Esto nos dice que quizás es una prueba poco exigente. Por tanto, teniendo en cuenta que lo costoso es ordenar los objetos para aplicar la estrategia voraz, vamos a aumentar progresiva y significativamente el número de objetos. Para ello haremos que la función reciba el tamaño y la llamaremos con un bucle que comience en el caso ya estudiado avanzando con un paso no excesivamente grande para el tamaño del intervalo.

$$N \in [300.000, 3.000.000] \quad \text{Paso} = 100.000 \quad M = 2 * N$$

La idea de este paso 'pequeño' es conseguir resultados muy precisos que nos permitan luego graficar.

Los resultados son los siguientes:



El fichero de esta prueba concreta se puede encontrar en [8]. Para ejecutarla solo es necesario hacer una llamada a la función *pruebaAleatoriosBucle* desde el *main* en el código que hay en [2].

Observamos un comportamiento muy bueno, tal y como esperábamos. Los valores analizados son bastante grandes y puede apreciarse el crecimiento asintótico de la función $n \cdot \log(n)$. Cuando este tamaño es muy grande el logaritmo comienza a tener influencia sobre la n y la gráfica deja de asemejarse a una función lineal.

2.2. Prueba sesgada favorable

Desde luego sesgando los datos podemos mejorar y empeorar casi a nuestro antojo este resultado 'promedio'.

El caso mejor, que abordamos a continuación consiste en que los datos venga ya ordenados según el criterio del algoritmo (v_i/p_i). Para ello vamos a repetir la prueba anterior pero con el vector previamente ordenado según este criterio, para no tenerlo en cuenta en el tiempo.

La salida obtenida es claramente mejor a la anterior (el coste es lineal). La prueba se puede encontrar en [2] y la salida concreta en [9].

La comparación gráfica con la anterior prueba es la que sigue:



Otro tipo de sesgo en los datos como diferencias en las capacidades de las mochilas o de los objetos que caben afectará en el bucle lineal, pero la ordenación, que es lo que más coste tiene, se hará igual, con un número de objetos n . Por ello no afectarán prácticamente en los resultados y no vale la pena estudiarlos. El coste en memoria apenas es el almacenaje de los datos de los objetos en un vector ($\in O(n)$) y por tanto tampoco nos vamos a parar mucho en analizarlo.

En este sentido, este algoritmo no es del todo comparable a los demás. Recordemos que resuelve un problema mucho más laxo que los que veremos a continuación donde si nos importarán más aspectos como la memoria y la naturaleza de los datos concretos del problema.

3. Algoritmo de programación dinámica

3.1. Prueba con tamaño de objetos y mochila variable

Para este algoritmo vamos a comenzar ejecutando los mismos test ya mencionados para el voraz. Tenemos que tener en cuenta que el coste es sensiblemente peor y por tanto no podremos alcanzar tamaños de entrada n tan grandes como antes, para mantener unos tiempos de espera razonables. El coste en tiempo teórico $\in O(n * M)$, de forma que ahora los parámetros a controlar en el estudio serán 2, ya que nos importa y perjudica que el tamaño de la mochila aumente, mientras que antes no.

Sin embargo, ya no tenemos que permitir fraccionabilidad de los objetos y podemos ser más exigentes imponiendo que, de escogerse, se metan enteros. Aunque si que es cierto que este algoritmo solo funciona con tamaño de la mochila y pesos $\in \mathbb{N}^+$, por lo que sigue estando algo limitado.

Para tamaños mucho más pequeños, la prueba aleatoria anterior arroja los siguientes resultados. Que pueden encontrarse en [5].



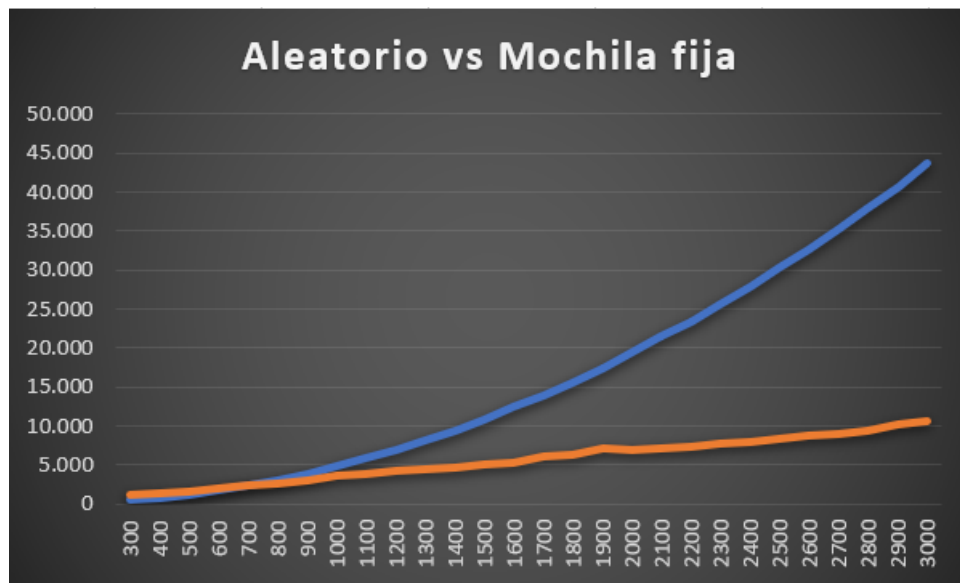
Como se puede apreciar en la gráfica, para casos de apenas unos miles de objetos ($n \in [300, 3.000]$), se han llegado a producir ejecuciones cercanas al minuto en las últimas iteraciones.

Claramente estamos ante un algoritmo incomparable al anterior. Trataremos de exprimirlo por otros medios.

3.2. Tamaño de mochila fijo

Como las variables que intervienen en el coste (n y M) se han hecho crecer aproximadamente al mismo ritmo, la gráfica se identifica inequívocamente con la parábola x^2 . Veamos que pasa si hacemos este aumento asimétrico.

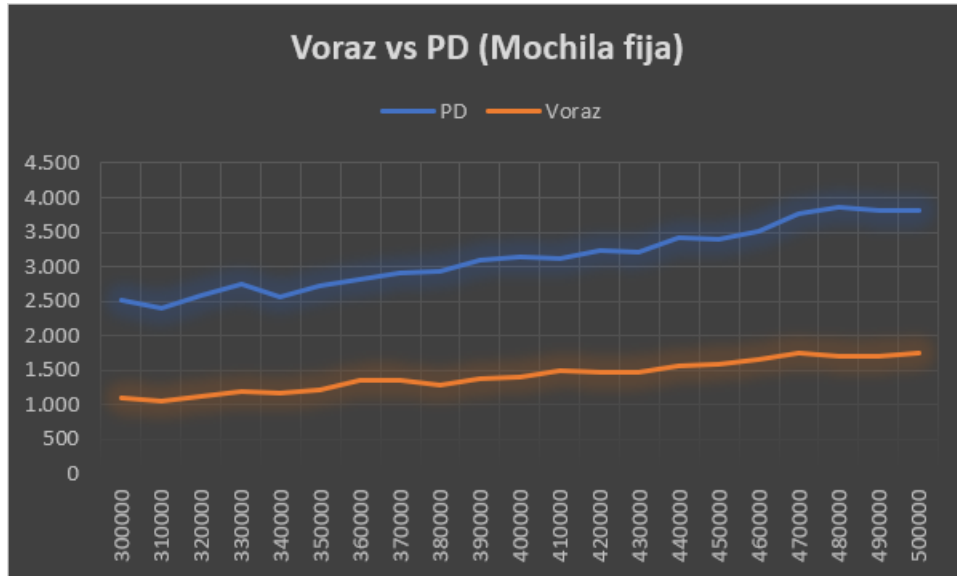
Comenzamos por considerar un tamaño de la mochila fijo y solo variable la cantidad de objetos. Para este tipo de problema nos encontraremos una coste sorprendentemente bueno. La M ejerce de constante y por tanto el algoritmo es asintóticamente lineal ($O(M*n)$). Pueden verse los resultados en [7].



Un hecho francamente notable es que teóricamente este algoritmo debería comportarse para casos suficientemente grandes mejor que el voraz. (Coste $O(M * n)$ frente a $O(n * \log(n))$).

Cuando $n \rightarrow \infty$, el tamaño de la mochila actúa como constante. Sin embargo para las implementaciones utilizadas no ocurre así. Veámoslo con un caso extremo en que debería ser mucho mejor, tamaños muy grandes y una mochila insignificante.

$$n \in [300.000, 500.000] \quad M = 1 \quad Paso = 10.000$$



Sobre el papel sería como enfrentar un algoritmo lineal, además de constante 1, frente a uno $\in O(n * \log(n))$. Pero los resultados no son los esperados.

Esto nos proporciona conclusiones interesantes. Los costes dados son respectivos al caso peor. Dado que en programación dinámica hay más accesos a memoria, que parecen ser más lentos y por la baja frecuencia con que se dan estos casos ‘malos’, resulta más interesante recurrir a la estrategia voraz siempre que sea posible.

Los resultados numéricos de la prueba se pueden encontrar en [6] y [10]. Mientras que la prueba se puede ejecutar con una simple llamada a *pruebaTamMochilaFijoBucle()* en [3] colocando previamente los tamaños y frecuencia de paso indicados.

3.3. Análisis de la pila de llamadas

Finalmente, vamos a ayudarnos de las herramientas que *Visual Studio* nos brinda para hacer un análisis en espacio de esta solución del problema. Nos resultará bastante útil para compararlo después con los algoritmos de vuelta atrás y de ramificación y poda, que darán mucho más juego en este sentido.

Para ello ejecutamos una única iteración del algoritmo con un caso suficientemente exigente como para que coste en memoria sea reseñable. Con $n = 5.000$ y $M = 10.000$ es necesario cerca de medio GB de memoria.

Tipo de objeto	Recuento	Tamaño (bytes) ▼
double[]	5.001	400.315.047
ProblemaMochila.exe!tObjeto[]	1	86.359
ProblemaMochila.exe!std::vector...	1	80.055
ProblemaMochila.exe!std::_Cont...	5.005	40.040
unsigned int[]	1	628
ProblemaMochila.exe!std::_Fac_n...	1	8

Figura 1: Datos de memoria

Como se puede apreciar en la Figura 1, tenemos un total de 5.001 vectores ($n = 5.000$) de tipo *double* que ocupan unos 400.000.000 de bytes en total. Si nos fijamos en cada uno de ellos, (Figura 2), encontramos una ocupación individual de 80.047 bytes cada uno que responde a los 8 bytes de cada *double* por las 10.001 posiciones debidas al tamaño de la mochila, más otros datos auxiliares que guarda el vector.

Instancia	Tamañ...	Antigüe... ▼
<0x1A058E08>	80.047	3905,557820
<0x1A045528>	80.047	3905,619153
<0x1A031C48>	80.047	3905,673820
<0x1A01E368>	80.047	3905,736486
<0x1A00AA88>	80.047	3905,803153
<0x19FF71A8>	80.047	3905,868486
<0x19FE38C8>	80.047	3905,932486
<0x19FCFFE8>	80.047	3906,002264

Figura 2: Tamaño de cada vector en memoria

En la práctica no será necesaria tanta precisión en los datos como para requerir el uso del tipo '*double*' que ocupa el doble que un real con tipo '*float*'. Con esta simple modificación en la implementación reducimos considerablemente el coste real en memoria. Sale de prácticamente la mitad (unos 250 MB) en el caso anterior de $n = 5.000$ como se puede apreciar en la Figura 3.

Tipo de objeto ▲	Recuento	Tamaño (bytes)
float[]	5.001	200.255.043
ProblemaMochila.exe!std::_Cont...	5.005	40.040
ProblemaMochila.exe!std::_Fac_n...	1	8
ProblemaMochila.exe!std::vector...	1	80.055
ProblemaMochila.exe!tObjeto[]	1	43.199
unsigned int[]	1	628

Figura 3: Datos de memoria con tipo float

4. Algoritmo de fuerza bruta

4.1. Prueba aleatorios

Como primera toma de contacto con algoritmos que exploren el árbol de soluciones asociado al problema de la mochila 0-1 veremos el de fuerza bruta. Se trata de un algoritmo trivial pero también muy ineficiente tanto en tiempo como en memoria, que por fin nos permite resolver el problema de meter los objetos enteros sin ningún tipo de hipótesis extra.

Los valores de todos los datos pueden ser reales cualesquiera. La idea es que como esta solución va a considerar todas las posibles combinaciones va a funcionar siempre, aunque veremos después varias maneras de mejorarla. El coste $\in O(2^n)$ en todos los casos ya que la exploración del árbol es completa independientemente de cómo sean los datos.

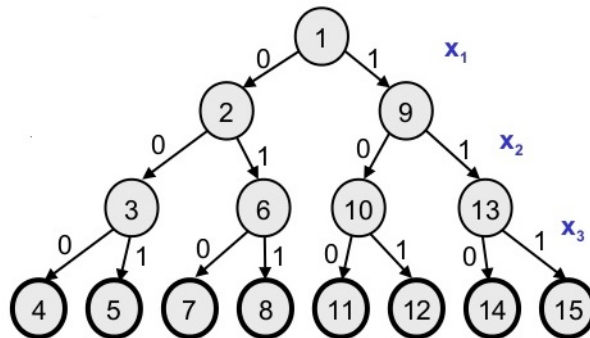
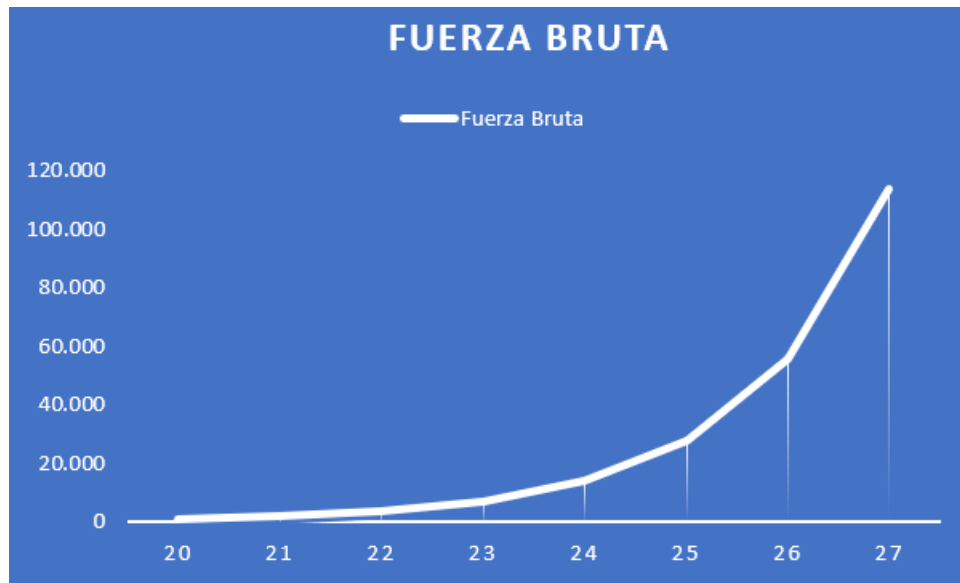


Figura 4: Árbol de exploración mochila 0-1

Ese coste desproporcionado, sobretodo en tiempo, no tarde en hacerse notar. Notamos unos terribles incrementos de espera a pequeños aumentos en el tamaño de objetos y apenas somos capaces, en periodos de tiempo razonables, de hacer iteraciones de unas pocas decenas de objetos. La prueba de aleatorios puede verse en [11] y se ha hecho con los siguientes datos:

$$n \in [20, 27] \quad PASO = 1 \quad M = 2 * n$$

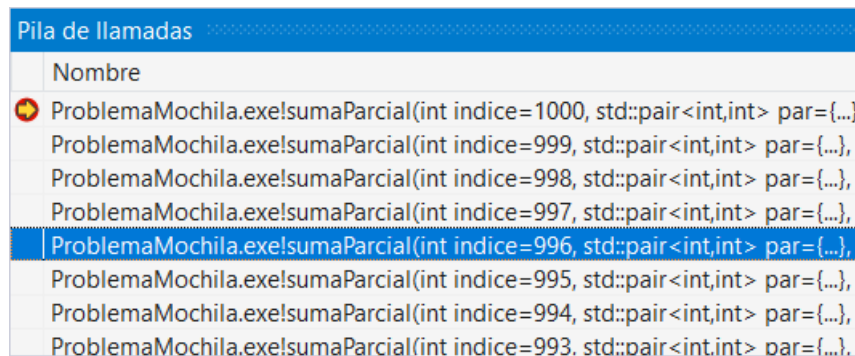


La gráfica no deja lugar a dudas, coste exponencial, con un crecimiento que para tamaño 20 apenas ronda el segundo (procesa $2^{20} = 1.048.576$ casos), mientras que está cerca de los 2 minutos para afrontar tan solo 7 casos más (con $n = 27$ procesa $2^{27} = 134.217.728$ casos). Esto es completamente lógico teniendo en cuenta que son casi 100 millones de casos más.

4.2. Análisis de memoria

Esta solución se presta muy bien a ser estudiada fijándonos en la pila de llamadas, pues es recursiva. En este caso no puede haber ninguna sorpresa y en un determinado momento de ejecución, habiendo llegado ya a una hoja del árbol, la pila debe tener tantas llamadas como tamaño de objetos hayamos introducido. Vamos a comprobarlo.

Para ello podemos probar con un caso más grande pues no necesitamos que el caso termine, sino simplemente ver la memoria que reserva la función al comenzar su ejecución. Pondremos aún así un tamaño razonable para que el programa no se aborte porque el tamaño de pila sea insuficiente. Elegimos $n = 1.000$ que está en la frontera de lo que *Visual Studio* acepta en este caso.



Pila de llamadas	
	Nombre
➡	ProblemaMochila.exe!sumaParcial(int indice=1000, std::pair<int,int> par={...})
	ProblemaMochila.exe!sumaParcial(int indice=999, std::pair<int,int> par={...})
	ProblemaMochila.exe!sumaParcial(int indice=998, std::pair<int,int> par={...})
	ProblemaMochila.exe!sumaParcial(int indice=997, std::pair<int,int> par={...})
	ProblemaMochila.exe!sumaParcial(int indice=996, std::pair<int,int> par={...})
	ProblemaMochila.exe!sumaParcial(int indice=995, std::pair<int,int> par={...})
	ProblemaMochila.exe!sumaParcial(int indice=994, std::pair<int,int> par={...})
	ProblemaMochila.exe!sumaParcial(int indice=993, std::pair<int,int> par={...})

Figura 5: Pila de llamadas en un momento de la iteración

El resultado es el esperado. Por otra parte el coste en memoria del sistema es paupérrimo ya que al ser la función recursiva y no necesitar estructuras auxiliares para acotar, apenas es necesario guardar el vector de objetos. Como se dijo para el algoritmo voraz, no vale la pena pararse a estudiarlo.

Nótese que para un problema bastante realista con 50 objetos ya sería absolutamente inviable esperar una respuesta del algoritmo. Y para tamaños ligeramente inferiores tendríamos que estar esperando horas. Es necesaria una solución mejor. Este algoritmo sirve como ejemplo de lo que no se debe hacer, caer siempre en el caso peor pudiendo evitarlo con ciertos cálculos mucho menos costosos.

5. Algoritmo de vuelta atrás

A continuación vamos a estudiar uno de los múltiples algoritmos que resuelven el problema mediante la técnica de *Backtracking*. Utilizaremos dos podas distintas sobre dicho algoritmo para evaluar cual es más interesante en el caso general. Podaremos por factibilidad y por optimalidad, y después con las dos a la vez. En la segunda utilizaremos el algoritmo voraz para calcular una cota dinámica, técnica que volveremos a utilizar en las estimaciones optimistas de ramificación y poda.

5.1. Poda por factibilidad

Para la primera poda hemos considerado una ordenación creciente de los objetos según su peso, de forma que cuando se esté profundizando en una rama y no se pueda meter uno, se descarten inmediatamente todos los de 'debajo' por tener necesariamente un peso mayor o igual.

Se trata de un poda cuya éxito depende mucho de la naturaleza de los datos a procesar. Para muestra, un botón. Si afrontamos un caso en que caben casi todos los objetos, no resultará nada efectiva. Para la prueba de aleatorios que veníamos haciendo con $n = 25$, y manteniendo un tamaño de la mochila de $2*n$ (caben casi todos los objetos), resulta incluso más lenta que la solución de fuerza bruta.

Tarda $\sim 70s$ frente a los $\sim 40s$ que tardaba la solución trivial. Simplemente reduciendo el tamaño de la mochila a la mitad ($M = n$), se reduce el tiempo de misma iteración a apenas 3 segundos. Podemos ver las salidas por consola de todo esto en las figuras 6 y 7.

```
N = 25
Tamano de la mochila 50
Nodos visitados 32636133
Tiempo requerido = 78.437 segundos
Presione una tecla para continuar . . .
```

Figura 6: Poda por factibilidad cuando caben muchos objetos con $n = 25$

El hecho de que, de un caso al otro hayamos visitado 32 millones de nodos menos, como hemos mostrado por pantalla, es muy revelador. En determinados casos una poda tan sencilla como esta puede ser muy útil.

```
N = 25
Tamano de la mochila 25
Nodos visitados 867623
Tiempo requerido = 3.371 segundos
Presione una tecla para continuar . . .
```

Figura 7: Poda por factibilidad cuando caben pocos objetos con $n = 25$

Sin embargo, esta cota no responde a aspectos puramente cuantitativos de la entrada, sino esencialmente cualitativos. Depende mucho de la ‘suerte’ que tengamos con el caso puede ser muy efectiva o no serlo. Ejecutando una caso para un mismo tamaño con objetos aleatorios, a veces nos encontramos tiempos muy buenos y otras veces no tanto. En la figura 8 podemos apreciar como a veces tenemos suerte y tarda poco y otras no tanta.

```
N = 25
Nodos visitados 5466979
Tiempo requerido = 17.765 segundos
-----
N = 26
Nodos visitados 2076405
Tiempo requerido = 6.848 segundos
-----
N = 27
Nodos visitados 4282787
Tiempo requerido = 14.08 segundos
-----
N = 28
Nodos visitados 12333699
Tiempo requerido = 39.823 segundos
-----
N = 29
Nodos visitados 27831415
Tiempo requerido = 89.661 segundos
-----
N = 30
Nodos visitados 17836755
Tiempo requerido = 58.553 segundos
-----
```

Figura 8: Poda por factibilidad cuando caben pocos objetos con $n = 25$

La implementación de esta solución con la mencionada cota y la prueba preparada para ejecutarse, ajustando las constantes de tamaños a los valores indicados, se pueden encontrar en [1].

5.2. Poda por optimalidad

Otra posible poda consiste en hallar de manera dinámica la posible mejor solución por la rama de exploración en curso. Así, descartamos seguir profundizando en la misma en caso de que esta cota sea inferior a la mejor solución encontrada. Esta cota se calculará asumiendo objetos fraccionables como ya vimos en el algoritmo voraz, de forma que resulte poco costosa aunque por ello no sea excesivamente fina. En principio será suficiente para apreciar una muy importante mejoría.

Los objetos serán previamente ordenados con el criterio que vimos en el voraz de forma que primero se exploran los más prometedores haciendo así más probable que las cotas se ajusten a la solución óptima no fraccionable por la rama y se pode en más casos.

```
N = 25
Nodos visitados 1906
Tiempo requerido = 0.014 segundos
-----
N = 26
Nodos visitados 6266
Tiempo requerido = 0.036 segundos
-----
N = 27
Nodos visitados 5928
Tiempo requerido = 0.045 segundos
-----
N = 28
Nodos visitados 46371
Tiempo requerido = 0.238 segundos
-----
N = 29
Nodos visitados 536054
Tiempo requerido = 2.045 segundos
-----
N = 30
Nodos visitados 5232396
Tiempo requerido = 18.818 segundos
-----
```

Figura 9: Poda por optimalidad para tamaños pequeños

Como se puede observar en la figura 9, para los tamaños anteriores tenemos tiempos mucho mejores que con la poda anterior. Aunque seguimos dependiendo de la fortuna en el caso a procesar. Esta claro que si se explora pronto la solución óptima o una cercana a la misma, todo lo que hubiera que mirar después se procesa en un abrir y cerrar de ojos. La implementación de la cota y la prueba pueden encontrarse en [1].

La eficiencia de la poda es altísima. Estamos descartando la exploración de prácticamente todo el árbol. Por ejemplo, para $n = 30$ donde tenemos $2^{30} = 1.073.741.824$ nodos en el árbol, apenas hemos necesitado visitar 5 millones. Es decir que nos hemos pulido más de mil millones de nodos simplemente siendo un poco astutos.

Reiteramos la importancia cualitativa del caso. El factor suerte en los aleatorios es determinante. Obsérvese la figura 10.

```
N = 25
Nodos visitados 5771
Tiempo requerido = 0.04 segundos
-----
N = 26
Nodos visitados 7092
Tiempo requerido = 0.057 segundos
-----
N = 27
Nodos visitados 9456
Tiempo requerido = 0.09 segundos
-----
N = 28
Nodos visitados 11701
Tiempo requerido = 0.165 segundos
-----
N = 29
Nodos visitados 12381
Tiempo requerido = 0.14 segundos
-----
N = 30
Nodos visitados 20355
Tiempo requerido = 0.189 segundos
-----
```

Figura 10: Caso especialmente bueno: poda por optimalidad con $M = n$

Para la ejecución de exactamente la misma prueba en la figura 10 podemos apreciar que se ha podado en todos los caso rapidísimo y ninguna iteración ha llegado siquiera al segundo de ejecución. Ha sido necesario mucho ejecutar la prueba para dar con tantos casos tan buenos, pero se demuestra que es posible que esto ocurra.

Sin embargo no siempre es así y la mayor parte de las veces nos encontramos unos casos muy rápidos y otros muy lentos de manera indistinta. Un ejemplo de ello es la salida que se puede ver en la figura 11

```
N = 25
Nodos visitados 221535
Tiempo requerido = 0.886 segundos
-----
N = 26
Nodos visitados 1305155
Tiempo requerido = 3.931 segundos
-----
N = 27
Nodos visitados 3148203
Tiempo requerido = 10.235 segundos
-----
N = 28
Nodos visitados 107477
Tiempo requerido = 0.346 segundos
-----
N = 29
Nodos visitados 106788
Tiempo requerido = 0.352 segundos
-----
N = 30
Nodos visitados 4385824
Tiempo requerido = 14.628 segundos
-----
```

Figura 11: Caso habitual poda por optimalidad

5.3. Combinación de podas y conclusiones

Finalmente vamos a ver que ocurre cuando aplicamos los dos mecanismos de poda a la vez. En principio nos permitirá evitar los casos ‘malos’ en que miramos gran parte de los nodos del árbol por más motivos, evitando casos especialmente lentos.

Efectivamente, los resultados son los esperados. Después de bastantes ejecuciones son muy pocos los casos en que sobrepasamos los 3 o 4 segundos en el rango $n \in [25, 30]$ en que nos estábamos moviendo. Sin embargo, no nos quitamos la fortuna de encima del todo y muy de vez en cuando aparecen casos en que hay que espera en torno a medio minuto o incluso más.

Si por lo que sea trabajamos con casos sesgados es posible que este algoritmo resultase tremendamente malo. Entonces sí que interesa recurrir a las primeras versiones que vimos (voraz y programación dinámica). Un caso suficientemente sesgado (cada solución es factible y mejor que la anterior en el orden de exploración) sería directamente no procesable para los tamaños que hemos estado estudiando. Con $n = 30$ que ronda los 1.000 millones de nodos podríamos estar horas.

6. Algoritmo de ramificación y poda

El último algoritmo que vamos a estudiar es primo hermano del que hemos visto en la sección anterior. Vamos a ver cómo se comporta la versión de ramificación y poda utilizando una cola de prioridad para visitar según lo ‘prometedores’ que son los nodos.

6.1. Poda Optimista-pesimista con el voraz

Utilizaremos como cota optimista la que nos da el voraz suponiendo objetos fraccionables como hemos hecho ya en la vuelta atrás y como cota pesimista la del voraz con objetos enteros. Es decir, que cuando un objeto nos toque meterlo fraccionado, para la optimista lo consideramos y para la pesimista no.

$$\text{Cota pesimista (Nodo X)} \leq \text{Valor real (Nodo X)} \leq \text{Cota Optimista (Nodo X)}$$

Los resultados ejecutando pruebas para tamaños muy pequeños para este algoritmo resultan algo decepcionantes. Se comporta bastante peor que la poda por optimalidad con vuelta atrás e incluso que cuando solo podábamos por factibilidad. Ahondando un poco en nuestro análisis encontramos una respuesta bien sencilla, las operaciones con la cola de prioridad son muy costosas y ralentizan bastante la ejecución.

La clave está en la salida que nos informa de los nodos visitados. La poda es eficaz. De hecho no tendría sentido que no lo fuera pues es una ampliación de la que ya nos dio muy buen resultado con vuelta atrás. La cantidad de nodos visitados es similar e incluso inferior que antes. Sin embargo las ejecuciones son más lentas. Cada nodo tarda mucho más en procesarse.

```
N = 20
Nodos visitados: 148429
Tiempo requerido = 29.445 segundos
-----
N = 21
Nodos visitados: 217118
Tiempo requerido = 43.428 segundos
-----
N = 22
Nodos visitados: 63963
Tiempo requerido = 12.623 segundos
-----
N = 23
Nodos visitados: 43664
Tiempo requerido = 7.847 segundos
-----
N = 24
Nodos visitados: 385471
Tiempo requerido = 92.714 segundos
-----
N = 25
Nodos visitados: 2120743
Tiempo requerido = 524.975 segundos
-----
```

Figura 12: Caso habitual poda por optimalidad

Como puede observarse en la figura 13, para un caso de apenas 63.000 nodos visitados se requiere de 12 segundos, caso que en vuelta atrás no habría durado ni un segundo. Más claro se ve con $n = 25$ donde ha sido necesario explorar unos 2 millones de nodos tardándose varios minutos. En el algoritmo por *Backtracking* tuvimos un caso con 4 millones de nodos visitados que se resolvió en apenas 12 segundos.

Si bien es cierto que descartaremos bastante nodos siempre. Un caso muy malo es más raro que nos afecte, por cómo miramos los nodos (siempre el más prometedor). Ahí no está el déficit de esta poda.

6.2. Coste en memoria

En este caso el espacio que ocupamos no es baladí. La cola de prioridad puede llegar a almacenar muchísimos elementos. En los casos que hemos visto anteriormente, no todos los nodos visitados tenían por que haber estado a la vez en la cola (metemos y sacamos), pero sí un número no muy lejano. Y apenas teníamos $n = 25$.

Aunque sea un caso artificial que va a tardar una eternidad en procesarse vamos a meter $n = 5.000$ que es el caso con el que hemos estudiado la memoria para el algoritmo de programación dinámica. El otro que utilizaba bastante memoria.

Observamos que la utilización de memoria va variando (entran y salen nodos de la cola) a diferencia de los casos en programación dinámica donde se mantenía constante ya que se reserva un determinado tamaño para la matriz durante toda la ejecución. Tras más de media hora de ejecución hemos llegado a la nada menos-preciable cantidad de 1 GB. Si lo hubiésemos prolongado probablemente estaríamos hablando de bastante más.

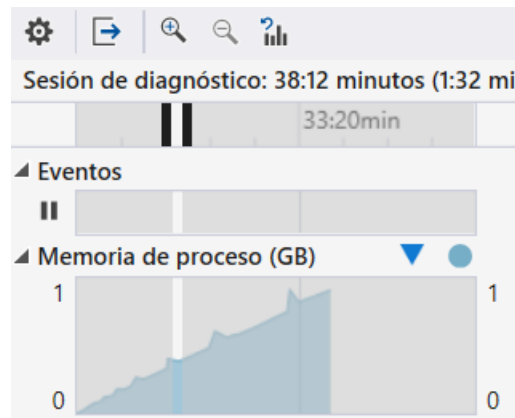


Figura 13: Memoria utilizada tras 30 minutos para $n = 5.000$

7. Comparativas finales

Para acabar, vamos a hacer una breve recapitulación de lo más importante entre todo lo que hemos visto con cierta perspectiva global.

Digamos que tenemos 2 bloques de algoritmos. Los que resuelven el problema en sí y los que resuelven uno más relajado. Los primeros son más lentos y consisten en vuelta atrás o ramificación y poda. Entre ellos hemos visto que, con cota de optimalidad, el de vuelta atrás es el más interesante. Pero que en un caso suficientemente malo podría resultar inaceptable mientras que el de ramificación y poda por cómo escoge los nodos tardaría también mucho, pero podría ser preferible.

Siempre que sea posible porque se cumplan las respectivas hipótesis extra acudiremos al voraz o al de programación dinámica. El primero es muy eficiente y nos ha permitido trabajar con tamaños de entrada altísimos y el segundo también es muy aceptable pidiendo hipótesis menos fuertes aunque es bastante costoso en memoria.

La siguiente tabla recoge los aspectos más relevantes:

Comparativa de algoritmos			
Algoritmo	Tiempo	Memoria	Problema
<u>Voraz</u>	Muy rápido	Lineal	Fraccionable
<u>Prog. Dinámica</u>	Rápido	$\in O(n * M)$	Datos $\in \mathbb{N}$
<u>Fuerza bruta</u>	Súper lento	Poca (pila)	Entero
<u>V. Atrás Fact</u>	Muy Lento	Poca (pila)	Entero
<u>V. Atrás Opt</u>	Lento	Poca (pila)	Entero
<u>Ramif y poda</u>	Muy lento	Bastante (Colas)	Entero

Referencias

- [1] Algoritmo y pruebas de vuelta atrás con varias cotas. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/MochilaVueltaAtras.cpp>,.
- [2] Fichero con las pruebas del método voraz. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/MochilaVoraz.cpp>.
- [3] Implementacion y pruebas con programación dinámica. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/MochilaPD.cpp>.
- [4] Implementación voraz de la solución al problema de la mochila. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/MochilaVorazImplementacion.cpp>.
- [5] Salida aleatorios programación dinámica. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/SalidaAleatoriosPD.txt>.
- [6] Salida programación dinámica mochila fija. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/SalidaMochilaFijaPD.txt>,.
- [7] Salida programación dinámica mochila fija de tamaño pequeño. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/SalidaMochilaFijaTamPequenoPD.txt>.
- [8] Salida prueba aleatoria método voraz. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/SalidaAleatorios.txt>. Salida por coordenadas de la gráfica en el mismo repertorio.
- [9] Salida prueba favorable método voraz. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/SalidaAleatoriosSesgadosBien.txt>.
- [10] Salida voraz con mochila fija. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/SalidaMochilaFijaVoraz.txt>,.
- [11] Solución y prueba por fuerza bruta. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/FuerzaBruta.cpp>,.