



UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

MÉTODOS ALGORÍTMICOS EN RESOLUCIÓN DE PROBLEMAS

Comparación de algoritmos. Problema de la mochila.

Autor:

Jorge Villarrubia Elvira

Student Number:

XXXXXX000

5 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
6. Algoritmo de ramificación y poda	13
7. Comparativas finales	13
8. Conclusiones	13

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 sesgados, 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 [3]. 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 u 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 el 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.

Donde 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 [1].

Observamos un comportamiento muy bueno, tal y como esperábamos. Los valores analizados son bastante grandes como para 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 peor 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 [1] y la salida concreta en [9].

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



Nótese que otro tipo de sesgo en los datos como diferencias en las capacidades de las mochilas o de los objetos que caben afectarán 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, antes no.

Por contra, 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 [4].



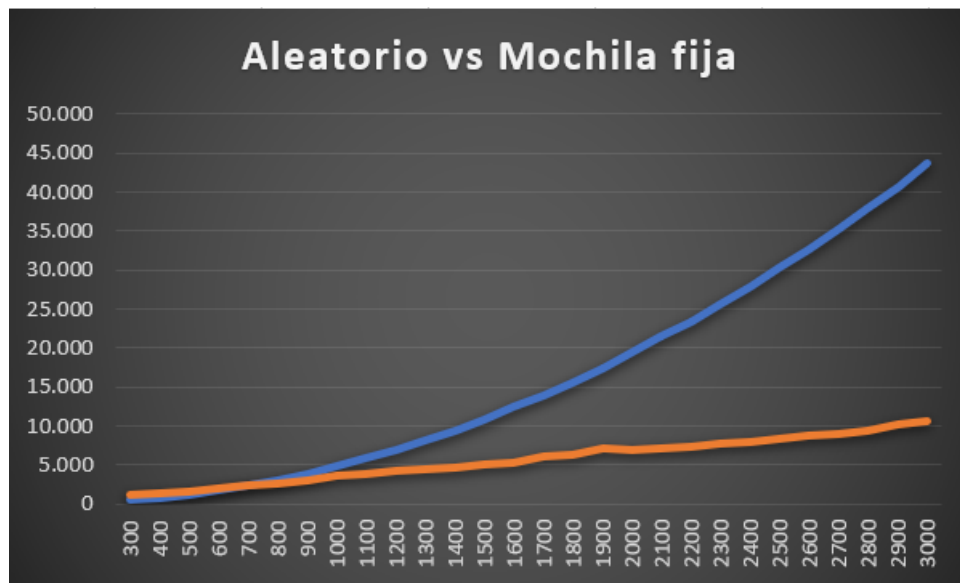
Como se puede apreciar en la gráfica, para casos mucho más pequeños que antes ($\in [300, 3.000]$), se han llegado a producir ejecuciones cercanas al minuto en las últimas iteraciones.

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.

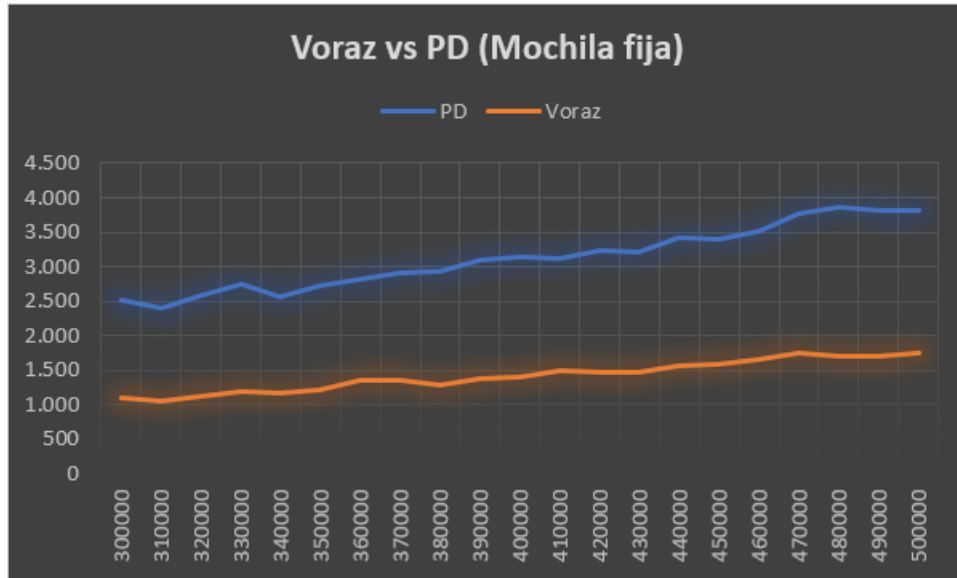
Comenzamos por considerar un tamaño de la mochila fijo y solo variable la cantidad de objetos. Para este tipo de problema nos encontraremos una coste sorprendentemente bueno. La M ejerce de constante y por tanto el algoritmo es asintóticamente lineal ($O(M*n)$). Pueden verse los resultados en [5].



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 dadas 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 [7]. Mientras que la prueba se puede ejecutar con una simple llamada a *pruebaTamMochilaFijoBucle()* en [2] 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 un compilador como *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.

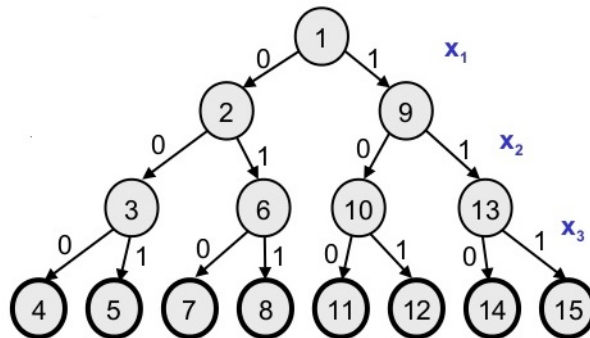


Figura 4: Árbol de exploración mochila 0-1

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 [10] y se ha hecho con los siguientes datos:

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

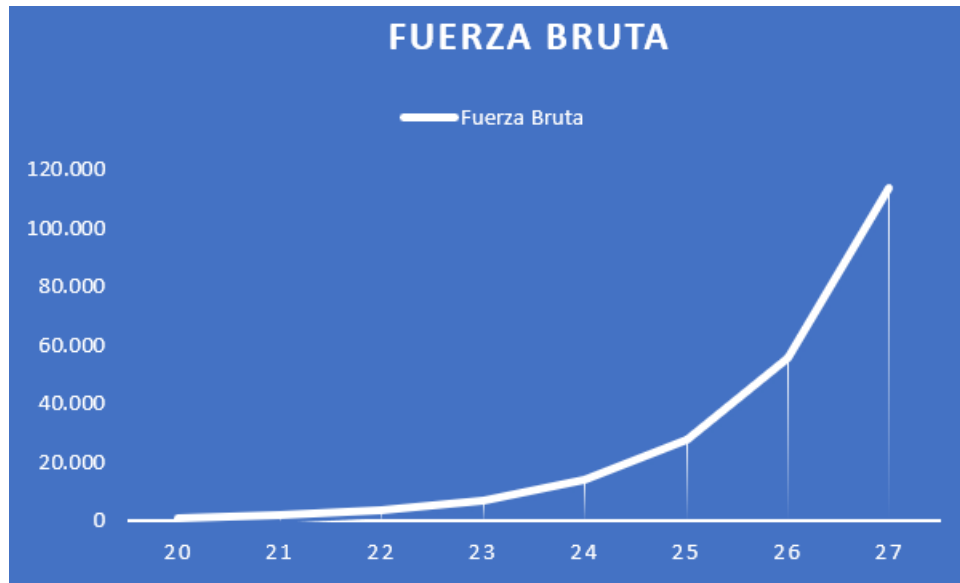


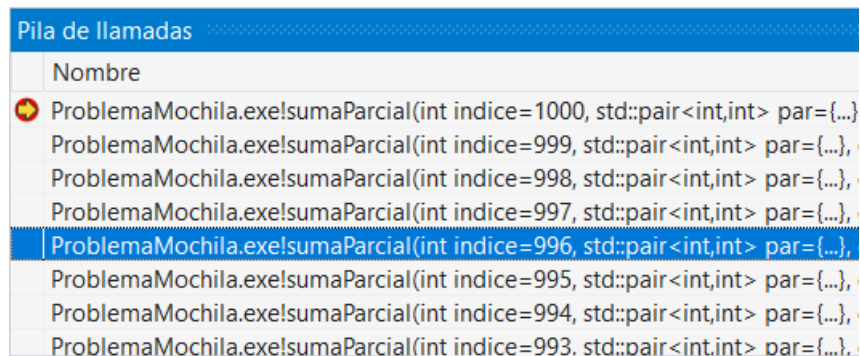
Figura 5: Coste en tiempo del algoritmo de fuerza bruta

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 y las que veremos después se prestan muy bien a ser estudiadas fijándonos en la pila de llamadas, pues son recursivas. En este caso no puede haber ninguna sorpresa y en un determinado momento de ejecución debe tener en la pila tantas llamadas como tamaño de objetos hayamos introducido. Cuando hagamos podas, esto nos permitirá saber de una forma rápida que zonas del árbol no esta siendo necesario visitar, dándonos idea de la eficacia de nuestra cota.

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 la 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 6: 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 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 problema. 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
6. Algoritmo de ramificación y poda
7. Comparativas finales
8. Conclusiones

— Linus

Referencias

- [1] Fichero con las pruebas del método voraz. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/MochilaVoraz.cpp>.
- [2] Implementacion y pruebas con programación dinámica. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/MochilaPD.cpp>.
- [3] Implementación voraz de la solución al problema de la mochila. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/MochilaVorazImplementacion.cpp>.
- [4] Salida aleatorios programación dinámica. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/SalidaAleatoriosPD.txt>.
- [5] Salida programación dinámica mochila fija. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/SalidaMochilaFijaTamPequenoPD.txt>.
- [6] Salida programación dinámica mochila fija tamaño. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/SalidaMochilaFijaPD.txt>,.
- [7] Salida programación dinámica mochila fija tamaño. <https://github.com/Jorgitou98/Pruebas-Problema-de-la-Mochila/blob/master/SalidaMochilaFijaVoraz.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] Solución y prueba por fuerza bruta.