DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Comparación de distintas técnicas algorítmicas para la resolución del *knapsack problem*

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2018

Integrante	LU	Correo electrónico
Berríos Verboven, Nicolas	46/12	nbverboven@gmail.com

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

Sea $X := \{x_0, x_1, \ldots, x_{n-1}\}$ un conjunto cuyos elementos tienen asociado un peso w_i y una ganancia v_i, con $1 \leq i < n$, y sea W un valor llamado capacidad máxima. Puede definirse un subconjunto $S \subseteq \{0, 1, ..., n-1\}$ de índices de X tal que la sumatoria de los pesos individuales de los elementos correspondientes en X sea menores o igual que W; es decir, $\sum_{j \in S} w_j \leq W$. El problema de la mochila (knapsack problem en inglés) consiste en encontrar un S tal que la suma de las ganancias de los elementos de X dados por los índices contenidos en S sea máxima,

Para la suma de pesos, podría considerarse el caso en que fuera posible agregar una fracción de un elemento, en cuyo caso añadiría una parte proporcional del valor original a la solución. Este trabajo, sin embargo, se basó en el supuesto de que, para un determinado elemento de X, las únicas posibilidades son agregarlo a la solución o no hacerlo. Esta versión del problema se conoce en inglés como *0/1 knapsack problem*.

Se definió una solución del problema como la sumatoria de los valores de los elementos indicados por S, sin importar cuáles fueran estos elementos y la solución óptima como el máximo del conjunto de todas las soluciones posibles.

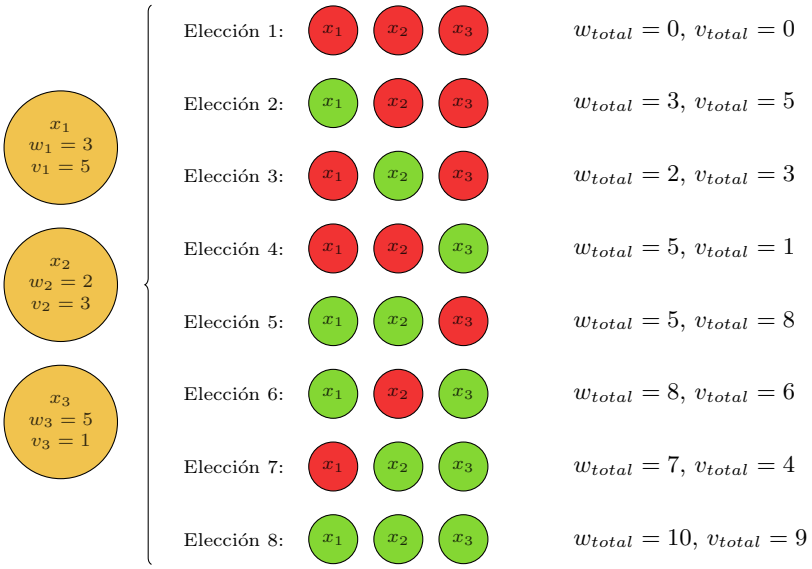

Figura 1: Ilustración de una parte de la resolución del problema de la mochila para un conjunto $X := \{x_1, x_2, x_3\}$. En el extremo izquierdo se muestran los elementos de X con sus respectivos pesos y ganancias. A su derecha, se listan todas las posibles combinaciones de estos, marcándose con rojo aquellos elementos que no se incluyen y con verde los que sí, junto con los pesos y las ganancias totales para cada elección. El que una de estas últimas sea solución o no del problema depende de la capacidad máxima W: para $W = 5$, las ganancias de las elecciones 1 a 5 son soluciones (dado que para todas ellas se cumple $w_{total} \leq W$) siendo la 5 la óptima. Si $W = 1$, la única opción posible es no elegir ninguno de los elementos de X y, por lo tanto, la solución óptima es 0 (asumiendo que este es el valor que se devuelve en este caso, pues podría tomarse otra decisión).

2. Desarrollo experimental

Para la implementación de los algoritmos se tomó la decisión de representar el conjunto de elementos a agregar con la clase `std::vector` de C++ y los elementos propiamente dichos mediante la clase `std::pair` utilizando un `int` para el peso y otro para el valor de un dado elemento. En el caso específico de este trabajo, los elementos son acciones con un costo (peso) y un retorno (valor). Para la escritura de los pseudocódigos que se presentan en las subsecciones siguientes se tuvo esto en consideración.

2.1. Algoritmo de fuerza bruta

Dado un vector de elementos A para el que se quiere resolver el problema de la mochila, se exploran todas las soluciones posibles (válidas o inválidas) recorriendo un árbol de decisión en donde cada nodo

(una solución parcial) tiene dos ramas, indicando si un elemento dado se considera o no, y las hojas corresponden a una solución construída en base al camino tomado para llegar a ellas. En cada paso se almacena el resultado parcial, tanto el costo como el retorno. En cuanto se llega a una hoja, se evalúa si la solución alcanzada es válida, en cuyo caso se compara la suma del retorno de sus elementos con el máximo encontrado hasta el momento y, si correspondiese, se actualiza dicho valor.

2.1.1. Correctitud del algoritmo

Algoritmo 1 Fuerza bruta

Input: A: vector $(a_0, a_1, \ldots, a_{n-1})$ de pares (w_i, v_i), donde w_i y v_i son el costo y el retorno del i-ésimo elemento de A respectivamente.
W: la capacidad máxima.
Output: Máximo $\sum_{i \in I} v_i$ tal que $\sum_{i \in I} w_i \leq W$, $I \subseteq \{0, 1, ..., n-1\}$.

```
1: función SOLUCIONFB(A, W)
2:     k ← 0                                  ▷ O(1) // Índice que marca el nivel actual en el árbol de decisión
3:     max ← 0   ▷ O(1) // Máximo retorno calculado entre todas las soluciones válidas exploradas
4:     cp ← 0                                 ▷ O(1) // Costo de la solución parcial
5:     rp ← 0                                 ▷ O(1) // Retorno de la solución parcial
6:     SOLUCIÓNFBAUX(A, W, max, cp, rp, k)
7:     devolver max
8: fin función

9: función SOLUCIONFBAUX(A, W, max, cp, rp, k)
10:    si k = |A| entonces                                                    ▷ O(1)
11:        si cp ≤ W ∧ rp > max entonces                                       ▷ O(1)
12:            max ← rp                                                       ▷ O(1)
13:        fin si
14:    si no
15:        solucionFBaux(A, W, max, cp + w_k, rp + v_k, k + 1)
16:        solucionFBaux(A, W, max, cp, rp, k + 1)
17:    fin si
18: fin función
```

El algoritmo comienza a recorrer el árbol desde la raíz. En cada paso, se hacen dos llamados recursivos: uno en donde el k-ésimo elemento del vector se selecciona y otro en donde no. Por cada una de estas decisiones, se realiza lo mismo con el $k+1$-ésimo elemento hasta llegar a $k = n = |A|$, luego de lo cual se finaliza. Es decir, en las hojas se obtienen todos los vectores $Z_i = (z_0, z_1, \ldots, z_{n-1})$, donde $0 \leq i < 2^n$, tales que z_j es 1 si se elige dicho elemento y 0 en caso contrario, con $0 \leq j < n$. De esta manera, es posible llegar a todas las hojas del árbol de decisión y, por lo tanto, se exploran todas las posibles soluciones.

Luego de comprobar que todas las combinaciones de elementos son generadas, resta ver que, al finalizar el recorrido, el valor de max corresponde efectivamente a una solución correcta del problema.

Si estoy en el caso base (es decir, llegué a alguno de los Z_i), veo si la suma del costo de los elementos no supera el valor máximo permitido; esto es, verifico que se trate de una solución válida. Si es así, actualizo el máximo con la suma de los retornos solamente si esta resulta estrictamente mayor que el valor previamente almacenado. De esta manera, me aseguro de guardar siempre el máximo.

Si estoy en un nodo de un nivel $k < |A|$, debo comprobar que la solución parcial se actualiza adecuadamente. Como puede verse en el algoritmo 1, cuando se llama recursivamente a SOLUCIONFBAUX agregando un elemento a la solución parcial, se suman su costo y su retorno a los correspondientes acumuladores mientras que, cuando esta llamada se realiza descartando el elemento, esto no ocurre. Por lo tanto, para cada solución parcial, los valores almacenados se modifican correctamente.

Si no fuera posible elegir ningún elemento, entonces se debería devolver 0 (el valor con el que se inicializa el máximo). Esto ocurre porque, en este caso, la única solución válida se obtiene tomando en cada nodo la rama negativa y esto trae como consecuencia que no se modifique el valor original.

Luego, el algoritmo propuesto resuelve el problema de la mochila.

2.1.2. Análisis de la complejidad temporal

La función principal SOLUCIONFB únicamente realiza asignaciones con costo $O(1)$ y un llamado a una función auxiliar, que será la que efectivamente resuelva el problema. Por lo tanto, la complejidad del algoritmo estará determinada por la de esta última.

Al comienzo de cada llamado a SOLUCIONFBAUX se realiza una comparación por igualdad entre k y $|A|$. Si además me encuentro en el caso base y debo actualizar el máximo se agregan un o lógico, dos comparaciones (por $=$ y $<$) y una asignación. Si, por el contrario, se trata de un paso intermedio, se realizan cuatro asignaciones y seis operaciones algebraicas (correspondientes a actualizar las soluciones parciales y el nivel k).

Asumiendo que estas operaciones se ejecutan en un tiempo constante (modelo uniforme) y teniendo en cuenta que en cada nivel se realizan dos llamadas recursivas en las que el tamaño de la subinstancia [1] se reduce en 1, puede escribirse la función $T(n)$, que describe la complejidad temporal en el peor caso, como

$$T(n) = \begin{cases} O(1) & \text{si } n = 0, \\ 2T(n-1) + O(1) & \text{si } n > 0. \end{cases}$$

Como lo que busco es una cota superior de complejidad, reescribo la ecuación anterior como

$$T(n) = \begin{cases} c & \text{si } n = 0, \\ 2T(n-1) + c & \text{si } n > 0, \end{cases}$$

donde c representa el máximo entre el tiempo que insume resolver el problema de tamaño 0 y el requerido para calcular las soluciones parciales.

Expandiendo la recurrencia anterior, se tiene

$$\begin{aligned} T(n) = 2T(n-1) + c &= 2[2T(n-2) + c] + c = 4T(n-2) + 3c \\ &= 4[2T(n-3) + c] + 3c = 8T(n-3) + 7c \\ &= 8[2T(n-4) + c] + 7c = 16T(n-4) + 15c \\ &= \ldots = 2^i T(n-i) + (2^i - 1)c \\ &= \ldots = 2^n T(n-n) + (2^n - 1)c \\ &= 2^n T(0) + (2^n - 1)c = 2^n c + (2^n - 1)c \\ &= 2^{n+1} c - c. \end{aligned}$$

Propongo $T(n) \leq r2^n - c$ para todo $n \geq 0$, con r una constante positiva, e intento demostrarlo por inducción en n.

Primero, supongo que $T(n-1) \leq r2^{n-1} - c$, con $r > 0$. Reemplazando en la ecuación original, se tiene que

$$\begin{aligned} T(n) = 2T(n-1) + c &\overset{HI}{\leq} 2(r2^{n-1} - c) + c \\ &= 2r2^{n-1} - 2c + c \\ &= r2^n - c \end{aligned}$$

y puede verse que esto vale para cualquier $r > 0$. Ahora, me interesa ver si la desigualdad vale para $T(0)$; es decir, si $T(0) = c \leq r2^0 - c = r - c$. Tomando $r \geq 2c$, llego a que $T(n) \leq r2^n - c$ para todo $n \geq 0$. Como c es un número positivo, $T(n) \leq r2^n$ para todo $n \geq 0$ y, como $r > 0$, $T(n) \in O(2^n)$.

Luego, la complejidad de SOLUCIONFB es $4O(1) + O(2^n) = O(2^n)$.

2.2. Algoritmo de backtracking

Para la resolución del problema de la mochila por backtracking se utilizó una variante del algoritmo de fuerza bruta descripto en la sección anterior que consiste en podar el arbol generado por el mismo de la siguiente forma:

[1] Definido como la longitud del subvector comprendido entre las posiciones k y $|A| - 1$.

- **Poda por factibilidad.** En cada nodo del árbol de backtracking verifico que el costo parcial de tomar la rama afirmativa (agregar un elemento) no supere el capital máximo disponible. Si fuera así, la descarto.

2.2.1. Correctitud del algoritmo

Algoritmo 2 Backtracking con poda por factibilidad

Input: A: vector $(a_0, a_1, \ldots, a_{n-1})$ de pares (w_i, v_i), donde w_i y v_i son el costo y el retorno del i-ésimo elemento de V respectivamente.
W: la capacidad máxima.
Output: Máximo $\sum_{i \in I} v_i$ tal que $\sum_{i \in I} w_i \leq W$, $I \subseteq \{0, 1, ..., n-1\}$.

```
1:  función SOLUCIONBT(A, W)
2:      k ← 0                                  ▷ O(1) // Índice que marca el nivel actual en el árbol de decisión
3:      max ← 0   ▷ O(1) // Máximo retorno calculado entre todas las soluciones válidas exploradas
4:      cp ← 0                                 ▷ O(1) // Costo de la solución parcial
5:      rp ← 0                                 ▷ O(1) // Retorno de la solución parcial
6:      SOLUCIÓNBTAUX(A, W, max, cp, rp, k)
7:      devolver max
8:  fin función

9:  función SOLUCIONBTAUX(A, W, max, cp, rp, k)
10:     si k = |A| entonces                                                          ▷ O(1)
11:         si cp ≤ W ∧ rp > max entonces                                             ▷ O(1)
12:             max ← rp                                                             ▷ O(1)
13:         fin si
14:     si no
15:         si (cp + w_k) ≤ W entonces          ▷ Podo si no puedo agregar. // O(1)
16:             SOLUCIONBTAUX(A, W, max, cp + w_k, rp + v_k, k + 1)
17:         fin si
18:         SOLUCIONBTAUX(A, W, max, cp, rp, k + 1)
19:     fin si
20: fin función
```

En el peor caso (considerándose como tal aquel en donde nunca encuentro un elemento que no pueda agregar) el algoritmo se convierte en uno de fuerza bruta y puede asumirse correcto por la sección anterior. Si, por el contrario, llegara a un punto en que el agregar un elemento a la solución parcial me llevaría a una solución inviable, entonces se toma la rama negativa sin modificar la solución previa. Como los valores del costo y del retorno parcial se modifican solo cuando se toma la decisión de agregar un elemento, las soluciones parciales se construyen adecuadamente.

2.2.2. Análisis de la complejidad temporal

Con la poda por factibilidad lo único que se agrega es un if con una comparación que se realiza en $O(1)$ en la función recursiva, mientras que la principal se mantiene sin cambios. En el peor caso, la complejidad sigue siendo la del algoritmo de fuerza bruta, ya que a la cantidad de operaciones que se realizan en cada llamada recursiva se le agrega solamente un factor constante. Por lo tanto, la complejidad temporal del algoritmo de backtracking podando por factibilidad es $O(2^n)$.

2.3. Algoritmo de programación dinámica

Dado un vector A de pares $(costo, retorno)$ y una capacidad máxima W, este algoritmo recorre una matriz de $|A| + 1$ por $W + 1$ posiciones, donde las filas representan los distintos tamaños de los subproblemas y las columnas las capacidades máximas de los mismos. A medida que avanza, encuentra las soluciones óptimas para cada combinación de n y w, con $0 \leq n < |A|$ y $0 \leq w < W$, llegando finalmente a la solución óptima del problema original.

2.3.1. Correctitud del algoritmo

Algoritmo 3 Programación dinámica

Input: A: vector $(a_0, a_1, \ldots, a_{n-1})$ de pares (w_i, v_i), donde w_i y v_i son el costo y el retorno del i-ésimo elemento de A respectivamente. Defino $n = |A|$.
W: la capacidad máxima.
Output: Máximo $\sum_{i \in I} v_i$ tal que $\sum_{i \in I} w_i \leq W$, $I \subseteq \{0, 1, ..., n-1\}$.

```
1: función SOLUCIONPD(A, W)
2:     m ← matriz de |A| + 1 filas y W + 1 columnas                ▷ O(|A|W)
3:     para i ← 0 hasta |W| + 1 hacer                              ▷ O(W)
4:         m_0i ← 0                                                ▷ O(1)
5:     fin para
6:     para i ← 1 hasta |A| + 1 hacer                              ▷ O(|A|)
7:         para j ← 0 hasta |W| + 1 hacer                          ▷ O(W)
8:             si w_{i-1} > j entonces                             ▷ O(1)
9:                 m_ij ← m_{(i-1)j}                               ▷ O(1)
10:            si no
11:                m_ij ← max(m_{(i-1)j}, v_i + m_{(i-1)(j-w_i)})  ▷ O(1)
12:            fin si
13:        fin para
14:    fin para
15:    devolver m_{|A|W}
16: fin función
```

Sean W una capacidad máxima y $A = (a_0, a_1, \ldots, a_{n-1})$ un vector de pares (w_i, v_i), donde w_i es el costo del i-ésimo elemento y v_i es su retorno, con $0 \leq i < n$.

Para comenzar, es necesario ver que se puede resolver el problema de la mochila mediante programación dinámica, y esto implica demostrar que se cumple el principio de optimalidad de Bellman.

Sea $S = \{s_0, s_1, \ldots, s_{m-1}\}$ un conjunto de índices de A tal que $W_S = \sum_{i \in S} w_i \leq W$ y maximiza a $V_S = \sum_{i \in S} v_i$; es decir, V_S es una solución óptima para el problema de la mochila dados A y W. Considerando que todos los elementos de S corresponden a elementos de A, para el índice $n-1$ existen dos posibilidades:

- $n-1 \notin S$. En este caso, $S \subseteq \{0, 1, ..., n-2\}$. Considerando el subvector $A' = (a_0, a_1, \ldots, a_n - 2)$, el máximo $\sum_{i \in S} v_i$ tal que $\sum_{i \in S} w_i \leq W$ debe ser V_S. Si no, existiría un conjunto S' de índices de A' tales que $W_{S'} = W_S \leq W$ y $V_{S'} > V_S$, y esto último es absurdo si se considera que V_S es la solución óptima. Entonces, V_S es una solución óptima para el subproblema de maximizar $\sum_{i \in S'} v_i$ sujeta a W_S para los elementos $a_0, a_1, ..., a_{n-2}$.
- $n-1 \in S$. Si ocurriera esto, entonces $V_S = v_{n-1} + V_{S'}$, donde S' es un conjunto de índices de $A' = (a_0, a_1, ..., a_{n-2})$ tal que $W_{S'} = W_S - w_{n-1}$ y $V_{S'} = V_S - v_{n-1}$ es máximo. Si no fuera así, podría definir un conjunto S'' de índices de A' tal que $W_{S''} = W_S - w_{n-1}$ y que $V_{S''} > V_{S'}$. Si a S'' le agrego $n-1$, sus costos siguen siendo menores o iguales que W (porque $W_{S''} + w_{n-1} = W_S \leq W$) pero ahora tengo que $V_{S''} = V_S + v_{n-1}$, lo que es absurdo ya que no puedo tener una solución mejor que la óptima. Entonces, $V_S = v_{n-1} + V_{S'}$, donde $V_{S'}$ es la solución óptima del subproblema de de maximizar $\sum_{i \in S'} v_i$ sujeta a $W_S - w_{n-1}$ para los elementos $a_0, a_1, ..., a_{n-2}$.

Habiendo caracterizado la solución óptima V_S, defino

$$f(i, w) = \text{Máxima} \sum_{j \in S \subseteq \{0,1,\ldots,i-1\}} v_j \text{ tal que } \sum_{j \in S \subseteq \{0,1,\ldots,i-1\}} w_j = w \text{ o } 0 \text{ si no existe,}$$

$$f(i, w) = \begin{cases} 0 & \text{si } i = 0, \\ f(i-1, w) & \text{si } i > 0 \text{ y } w_{i-1} > w, \\ max(f(i-1, w), v_{i-1} + f(i-1, w - w_{i-1})) & \text{si } i > 0 \text{ y } w_{i-1} \leq w. \end{cases}$$

Entonces, el problema puede ser resuelto mediante programación dinámica. El algoritmo 3 corresponde a una versión *bottom-up* de la función recursiva.

Para ver que no se calcula dos veces la misma solución, basta con observar que cada posición de la matriz se visita una única vez. Los valores de los casos base se inicializan en 0, de acuerdo con $f(i, w)$ y, para el resto de las posiciones, si $w_{i-1} > w$, entonces $m_{iw} = m_{(i-1)w}$, lo que corresponde a calcular $f(i-1, w)$. Si $w_{i-1} \leq w$, entonces $m_{iw} = max(m_{(i-1)w}, v_{i-1} + m_{(i-1)(w-w_{i-1})})$ que, nuevamente, se corresponde con el respectivo caso de f. El resultado final, luego de haber computado todos los posibles, equivale a obtener $f(|A|, W)$, es decir la solución óptima para el problema de tamaño $|A|$.

2.3.2. Análisis de la complejidad temporal

Lo primero que se hace es inicializar la matriz para almacenar los resultados, lo que acarrea un costo de $O(|A|W)$ considerando que las dimensiones son $|A| + 1$ filas y $W + 1$ columnas.

A continuación, se setean los valores de la fila correspondiente a $i = 0$ en 0, para representar el casos base. Esto se realiza con costo

$$\sum_{i=0}^{W} O(1) = (W+1)O(1) = O(W).$$

Por último, se recorre cada posición restante de la matriz aplicando los casos según lo indicado por la fórmula recursiva. Comenzando por la columna 0 y la fila 1, cada posición se visita una vez, realizando operaciónes con costo $O(1)$ en cada caso (estas son: comparaciones booleanas, accesos a posiciones de la matriz y sumas y restas). La sumatoria resulta

$$\sum_{i=0}^{|A|} \sum_{j=0}^{W} O(1) = (|A|+1)(W+1)O(1) = O(|A|W).$$

La complejidad temporal del algoritmo es la suma de todas estas, lo que, tomando $n = |A|$ nos da

$$2O(|A|W) + O(W) = O(|A|W) = O(nW).$$

2.4. Diseño de los experimentos

Los casos de prueba son los mismos utilizados por Horowitz y Sangi [1] con dos modificación que consisten en incluir los extremos de los intervalos de los que se toman los valores aleatorios y cambiar una de las capacidades máximas por un valor menor. Esto último se debe a que no fue posible observar diferencias con los W utilizados por los mencionados autores. A continuación se transcribe la lista como se encuentra en el trabajo, con las modificaciones mencionadas y renombrando los términos en castellano:

- costos al azar w y retornos al azar $v_i; 1 \leq w_i, v_i \leq 100$.
- costos al azar w y retornos al azar $v_i; 1 \leq w_i, v_i \leq 1000$.
- costos al azar w, $1 \leq w_i \leq 100, v_i = w_i + 10$.
- costos al azar w, $1 \leq w_i \leq 1000, v_i = w_i + 100$.
- retornos al azar v, $1 \leq v_i \leq 100, w_i = v_i + 10$.
- retornos al azar v, $1 \leq v_i \leq 1000, w_i = v_i + 100$.

Se utilizaron dos capacidades máximas dado un vector A, con $|A| = n$, de pares (w_i, v_i): $W_1 = 2\sum_{i=0}^{n-1} a_{i_1}$ y $W_2 = \frac{1}{5}\sum_{i=0}^{n-1} a_{i_1}$. Esto también fue tomado del trabajo mencionado anteriormente con la salvedad de que, en el caso de W_2, se dividió la suma de pesos por 5 en lugar de hacerlo por 2 como figuraba en el paper. Notar que en el primer caso todos los elementos podrán ser incluídos en la mochila, mientras que en el segundo necesariamente habrá algunos que quedarán fuera. Se espera que esto no afecte al tiempo de ejecución del algoritmo de fuerza bruta, dado que siempre prueba todas las posibilidades (todo caso sería el peor caso), y que sí lo haga con el de programación dinámica, porque su complejidad depende directamente de la capacidad máxima, y con el de backtracking . Para este

último, W_1 constituye un peor caso, puesto que ninguna rama podría ser podada, y esto hace pensar que los tiempos de los casos que usen esta capacidad máxima serán menores que aquellos de los que utilizen W_2, para el que sí sería posible efectuar podas.

Cada caso se corrió con los tres algoritmos para vectores de tamaño entre 1 y 28, con una granularidad de 1. Las pruebas para cada algoritmo se realizaron por separado utilizando el mismo vector para las dos capacidades máximas. Para cada tamaño se tomaron 25 muestras, calculándose luego el promedio entre ellas. Se tomó esta determinación en base a que se encontró que el valor promedio de la muestra se estabilizaba a partir de los 20 elementos. El rango utilizado responde a que en los ensayos preliminares se encontró que, el tiempo de ejecución tanto del algoritmo de fuerza bruta como del de backtracking (en el caso de no poder podar) era considerablemente superior al de programación dinámica, además de ser muy elevado. Se consideró que el rango era suficiente para el propósito de comparar los tres algoritmos y que, por lo tanto, no era justificado el incremento en el tiempo requerido para la realización de los experimentos.

3. Resultados y discusión

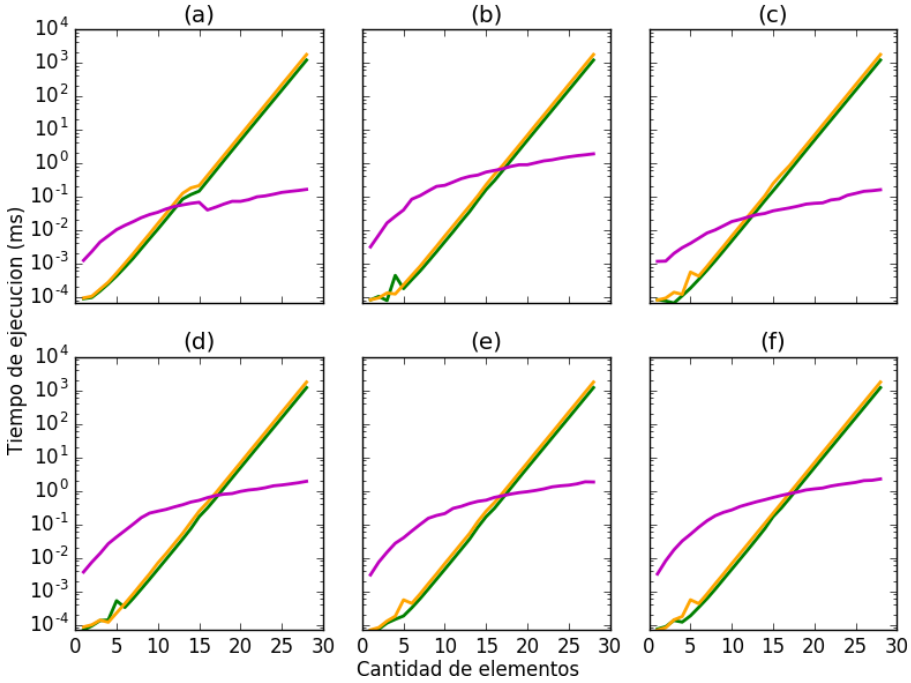

Figura 2: Tiempo de ejecución en función del tamaño de la entrada (graficados en escala logarítmica) de los algoritmos de fuerza bruta (/) , backtracking (/) y programación dinámica (/) para los casos: w_i, v_i al azar entre 0 y 100 (a) y 0 y 1000 (b); w_i al azar entre 0 y 100, $v_i = w_i + 10$ (c); w_i al azar entre 0 y 1000, $v_i = w_i + 100$ (d); v_i al azar entre 0 y 100, $w_i = v_i + 10$ (e); v_i al azar entre 0 y 1000, $w_i = v_i + 100$ (f). La capacidad máxima utilizada fue $W_1 = 2\sum_{i=0}^{n-1} a_{i_1}$. w_i y v_i son el costo y el retorno del i-ésimo elemento, respectivamente.

En las figuras 2 y 3 puede observarse que asintóticamente el algoritmo de programación dinámica se comporta mejor que los otros dos, lo que tiene sentido si se piensa que el primero es pseudopolinomial mientras que los otros son exponenciales. No obstante, para todos los tamaños hasta cierto valor (alrededor de los 15/20 elementos para W_1 y algo menos para W_2) los más rápidos resultan ser los algoritmos con complejidad más alta; esto se debe a que, para un vector con pocos elementos, el costo de generar la matriz de soluciones y recorrerla completa es mayor que el de generar todas las soluciones posibles. Esto puede apreciarse, por ejemplo, en las figuras 1a y 2a donde el tiempo de ejecución del algoritmo de programación dinámica disminuye levemente alrededor del punto en que comienza a desempeñar mejor que los otros.

Como se esperaba, el tiempo de ejecución del algoritmo de fuerza bruta no se vio modificado al cambiar la capacidad máxima. Esto se debe a que en todos los casos calcula las 2^n soluciones posibles, independientemente del valor de W.

Tanto el algoritmo de programación dinámica como el de backtracking mostraron diferencias al

ser corrido con diferentes W, siendo más rápidos cuando este era $\frac{1}{5}$ de la suma de los costos. Esto se debe, en el primer caso, a que el tamaño de la matriz de soluciones es menor y, por lo tanto, el tiempo requerido para su creación y recorrido es menor. En el segundo caso, esto es provocado porque, como necesariamente habrá elementos que no se podrán agregar a la solución, el algoritmo realiza podas, cosa que no ocurre en el peor caso que se muestra en la figura 2.

Si bien las podas mejoran el rendimiento del algoritmo de backtracking con respecto al de de fuerza bruta, su peor caso sigue siendo peor que el de este, lo cual se debe a que, en el peor caso, la cantidad de operaciones que realiza es mayor.

Para ninguno de los tres algoritmos se observó diferencias entre el tipo de elemento que contenía el arreglo.

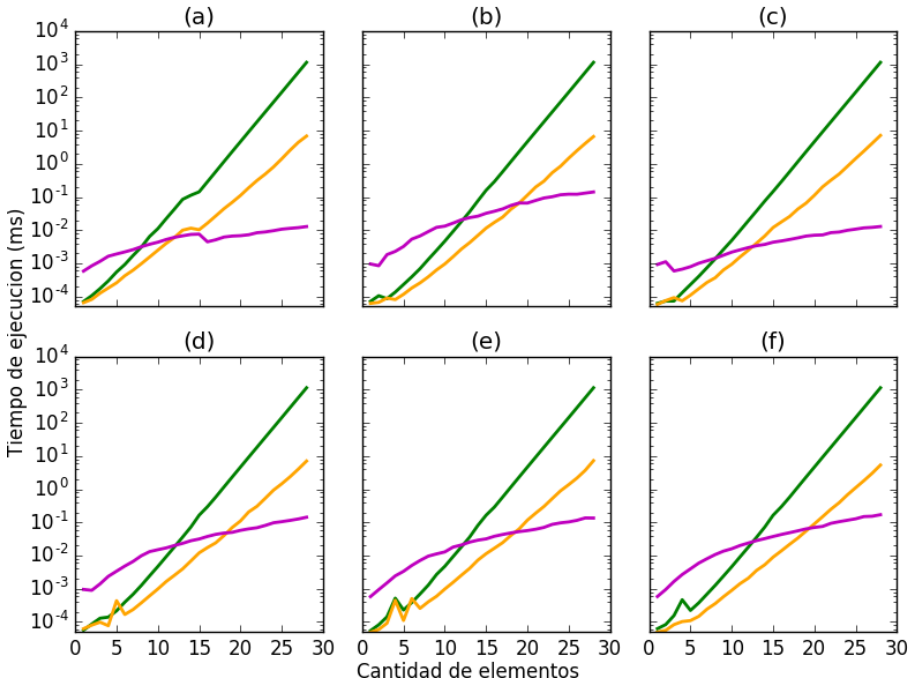

Figura 3: Tiempo de ejecución en función del tamaño de la entrada (graficados en escala logarítmica) de los algoritmos de fuerza bruta (/) , backtracking (/) y programación dinámica (/) para los casos: w_i, v_i al azar entre 0 y 100 (a) y 0 y 1000 (b); w_i al azar entre 0 y 100, $v_i = w_i + 10$ (c); w_i al azar entre 0 y 1000, $v_i = w_i + 100$ (d); v_i al azar entre 0 y 100, $w_i = v_i + 10$ (e); v_i al azar entre 0 y 1000, $w_i = v_i + 100$ (f). La capacidad máxima utilizada fue $W_2 = \frac{1}{5}\sum_{i=0}^{n-1} a_{i_1}$. w_i y v_i son el costo y el retorno del i-ésimo elemento, respectivamente.

4. Conclusiones

En este trabajo se comparó el tiempo de ejecución de tres algoritmos que resuelven el Problema de la mochila 0/1 mediante las técnicas de fuerza bruta, backtracking y programación dinámica. Se encontró que el tercero, con una complejidad temporal pseudopolinomial se desempeñó mejor que los otros (exponenciales). Esto exhibe la razón por la que, si bien en este caso particular no se conoce, siempre es deseable encontrar un algoritmo polinomial para un problema dado. Además, se observó que, si bien el backtracking permite mejorar (en promedio) un algoritmo de fuerza bruta, sigue resultando peor asintóticamente que el de programación dinámica.

A pesar de que la complejidad favorece al algoritmo de programación dinámica, para resolver el problema dada una cantidad pequeña de elementos resultaron más eficientes los método asintóticamente peores. Esto permite vislumbrar una forma de optimizar la resolución del problema, combinando algoritmos para maximizar la velocidad de ejecución (algo de éstas características ocurre com Timsort, el algoritmo de ordenamiento para listas que utiliza Python [2] por defecto).

Existen otros ensayos que, por una cuestión de tiempos no fue posible llevar a cabo. Entre ellos está la poda por optimalidad del árbol de backtracking y estudiar las diferencias (si existieran) con la poda por factibilidad y la combinación de ambas y comparar el algoritmo de programación dinámica

con uno optimizado para utilizar solo dos vectores en lugar de una matriz completa. Fuera de esto, se considera que se cumplieron los objetivos planteados para este trabajo prático.

Referencias

[1] Ellis Horowitz and Sartaj Sanhi, Computing Partitions with Applications to the Knapsack Problem, *J. ACM*, 21(2):277–292, April 1974.

[2] `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`