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 $S := \{x_1, x_2, \ldots, x_n\}$ un conjunto cuyos elementos tienen asociado un peso w_i y una ganancia v_i, con $1 \leq i \leq n$, y sea W un valor llamado capacidad máxima. Puede definirse un subconjunto $S' := \{y_1, y_2, \ldots, y_m\} \subseteq S$ tal que la sumatoria de los pesos individuales de sus elementos sean menores o iguales que W; es decir, $\sum_{j=0}^{m} y_j \leq W$. Otra forma de ver esto es tomando un vector $z = (z_1, z_2, \ldots, z_n)$, donde z_i, $1 \leq i \leq n$ es 1 o 0 dependiendo de si el elemento $x_i \in S$ se encuentra en S' o no. De esta manera, la sumatoria anterior también puede escribirse como $\sum_{i=0}^{n} z_i x_i$. El Problema de la mochila (knapsack problem en inglés) consiste en encontrar S' tal que la suma de las ganancias de sus elementos sea máxima.

Para la suma de pesos uno podría considerar el caso de 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 dado elemento de S, las únicas posibilidades son agregarlo a la solución o no hacerlo. Esta versión del problema se conoce en inglés como *0-1 knapsack problem*.

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 costo y otro para el retorno. Para la escritura de los pseudocódigos que se presentann en las subsecciones siguientes se tuvo esto en consideración.

2.1. Algoritmo de fuerza bruta

Dado un vector de elementos V 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: V: vector $(v_0, v_2, \ldots, v_{n-1})$ de pares (p_i, w_i), donde p_i y w_i son el costo y el retorno del i-ésimo elemento de V respectivamente.
W: la capacidad máxima.
k: índice que marca el nivel actual en el árbol de decisión.
max: el máximo retorno calculado entre todas las soluciones válidas exploradas.
cp y rp: respectivamente, el costo y el retorno de la solución parcial luego de la k-ésima llamada recursiva.
Output: max, donde max es el máximo $\sum_{i=0}^{n-1} v_{i_2}$ tal que $\sum_{i=0}^{n-1} v_{i_1} \leq W$.
Algoritmo: Inicializo $max, cp, rp, k \leftarrow 0$
solucionFB(V, W, max, cp, rp, k)

```
1: if k = |V| then                                              ▷ O(1)
2:     if cp ≤ W ∧ rp > max then                                ▷ O(1)
3:         max ← rp                                             ▷ O(1)
4:     end if
5: else
6:     solucionFB(V, W, max, cp + v_{k_1}, rp + v_{k_2}, k + 1)
7:     solucionFB(V, W, max, cp, rp, k + 1)
8: end if
```

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 $k = n = |V|$, luego de lo cual se finaliza. Es decir, en las hojas se obtienen todos los vectores $v_i = (v_0, v_1, \ldots, v_{n-1})$, donde, $0 \leq i < 2^n$, tales que v_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 v_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 $k < |V|$, debo comprobar que la solución parcial se actualiza adecuadamente. Como puede verse en el algoritmo 1, cuando se llama recursivamente a *solucionFB* 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

Al comienzo de cada llamado a *solucionFB* se realiza una comparación por igualdad entre k y $|V|$. 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 $m > 0$ tal que $m \leq n$ y, en particular, para $m = n - 1$, lo que implica que $T(n-1) \leq r2^{n-1} - c$ para alguna constante $r > 0$. Reemplazando en la ecuación original,

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

se tiene que

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

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 $r > 0$, $T(n) \in O(2^n)$.

Dado que $O(2^n) \subset O(n2^n)$, la complejidad del algoritmo propuesto cumple con el requerimiento del enunciado.

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:

- **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: V: vector $(v_0, v_2, \ldots, v_{n-1})$ de pares (p_i, w_i), donde p_i y w_i son el costo y el retorno del i-ésimo elemento de V respectivamente.
W: la capacidad máxima.
k: índice que marca el nivel actual en el árbol de decisión.
max: el máximo retorno calculado entre todas las soluciones válidas exploradas.
cp y rp: respectivamente, el costo y el retorno de la solución parcial luego de la k-ésima llamada recursiva.
Output: max, donde max es el máximo $\sum_{i=0}^{n-1} v_{i_2}$ tal que $\sum_{i=0}^{n-1} v_{i_1} \leq W$.
Algoritmo: Inicializo $max, cp, rp, k \leftarrow 0$
solucionBT(V, W, max, cp, rp, k)

```
1: if k = |V| then                                              ▷ O(1)
2:     if cp ≤ W ∧ rp > max then                                ▷ O(1)
3:         max ← rp                                             ▷ O(1)
4:     end if
5: else
6:     if (cp + v_{k_1}) ≤ W then          ▷ Podo si no puedo agregar. // O(1)
7:         solucionBT(V, W, max, cp + v_{k_1}, rp + v_{k_2}, k + 1)
8:     end if
9:     solucionBT(V, W, max, cp, rp, k + 1)
10: end if
```

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 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 $O(2^n) \subset O(n2^n) \subset O(n^2 2^n)$. Entonces, se cumple con lo solicitado en el enunciado.

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 $(a_0, a_2, \ldots, a_{n-1})$ 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. Defino $n = |A|$.
W: la capacidad máxima.
Output: $m_{|A|W}$, donde $m_{|A|W}$ es el máximo $\sum_{i=0}^{n-1} a_{i_2}$ tal que $\sum_{i=0}^{n-1} a_{i_1} \leq W$.
Algoritmo:
solucionPD(A, W)

```
1:  m ← matriz de |A| + 1 filas y W + 1 columnas                    ▷ O(|A|W)
2:  for i = 0 to |W| + 1 do                                          ▷ O(W)
3:      m_0i ← 0                                                     ▷ O(1)
4:  end for
5:  for i = 0 to |A| + 1 do                                          ▷ O(|A|)
6:      for j = 0 to |W| + 1 do                                      ▷ O(W)
7:          if a_{i-1_1} > W then                                    ▷ O(1)
8:              m_ij ← m_{i-1j}                                      ▷ O(1)
9:          else
10:             m_ij ← max(m_{i-1j}, a_{i_2} + m_{i-1j-a_{i_1}})     ▷ O(1)
11:         end if
12:     end for
13: end for
```

Sean W una capacidad máxima y $A = (a_0, a_2, \ldots, a_{n-1})$ 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 encontrar el máximo $\sum_{i=0}^{n-1} v_i$ sujeto a la restricción $\sum_{i=0}^{n-1} w_i \leq W$ mediante programación dinámica, y esto implica demostrar que se cumple el principio de optimalidad de Bellman.

Supongo que $S = \{s_0, s_1, \ldots, s_{m-1}\} \subseteq A$ es un conjunto de elementos de A tal que $W_S = \sum_{i=0}^{m-1} s_{i_1} \leq W$ maximiza $V_S = \sum_{i=0}^{m-1} s_{i_2}$. Considerando que todos los elementos de S pertenecen a A, para el elemento a_{n-1} existen dos posibilidades:

- $a_{n-1} \notin S$. En este caso, considerando el subvector $A' = (a_0, a_1, \ldots, a_n - 2)$, el máximo $\sum_{i=0}^{n-2} s_{i_2}$ tal que $\sum_{i=0}^{n-2} a_{i_1} \leq W$ debe ser V_S. Si no, existiría un conjunto $S' \subseteq A'$ de elementos de A' tales que $W_{S'} \leq W$ 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 $\sum_{i=0}^{n-2} s_{i_2}$ sujeta a W_S para los elementos $a_1, a_2, ..., a_{n-2}$.

- $a_{n-1} \in S$. En este caso, entonces $V_S = \{a_{n-1_2}\} \cup V_{S'}$, donde S' es un conjunto de elementos de $A' = (a_0, a_1, ..., a_{n-2})$ tal que $W_{S'} = W_S - a_{n-1_1}$ y $V_{S'} = V_S - a_{n-1_2}$ es máximo. Si no fuera así, podría definir un conjunto S'' de elementos de A' tal que $W_{S''} = W_S - a_{n-1_1}$ y que $V_{S''} > V_{S'}$. Si a S'' le agrego el elemento a_{n-1}, sus costos siguen siendo menores o iguales que W (porque $W_{S''} + a_{n-1_1} = W_S \leq W$) pero ahora tengo que $V_{S''} = V_S + a_{n-1_2}$, lo que es absurdo ya que no puedo tener una solución mejor que la óptima. Entonces, $V_S = \{a_{n-1_2}\} \cup V_{S'}$, donde $V_{S'}$ es la solución óptima del subproblema de de maximizar $\sum_{i=0}^{n-2} s_{i_2}$ sujeta a $W_S - a_{n-1_1}$ para los elementos $a_1, a_2, ..., a_{n-2}$.

Habiendo caracterizado la solución óptima S, defino

$$f(i,w) = \text{Máxima } \sum_{i=0}^{i-1} a_{i_2} \text{ tal que } \sum_{i=0}^{i-1} a_{i_1} = w \text{ o } 0 \text{ si no existe,}$$

$$f(i,w) = \begin{cases} 0 & \text{si } i = 0, \\ f(i-1,w) & \text{si } i > 0 \text{ y } a_{i-1_1} > w, \\ max(f(i-1,w), a_{i-1_2} + f(i-1, w - a_{i-1_1})) & \text{si } i > 0 \text{ y } a_{i-1_1} \leq w. \end{cases}$$

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 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 $a_{i-1_1} > w$, entonces $m_{i,w} = m_{i-1,w}$, lo que corresponde a calcular $f(i-1,w)$. Si $a_{i-1_1} \leq w$, entonces $m_{i,w} = max(m_{i-1,w}, a_{i-1_2} + m_{i-1,w-a_{i-1_1}})$ 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

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

que cumple con lo que se requería en el enunciado.

2.4. Diseño de los experimentos

Los casos de prueba son los mismos utilizados por Horowitz y Sangi [1] con una modificación que consiste en incluir los extremos de los intervalos de los que se toman los valores aleatorios. 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 $W_2 = \frac{1}{2}\sum_{i=0}^{n-1} a_{i_1}$. Esto también fue tomado del trabajo mencionado anteriormente.

Cada caso se corrió con los tres algoritmos para vectores de tamaño entre 1 y 30, 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 35 muestras, calculándose luego el promedio entre ellas. 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 era considerablemente superior al de programación dinámica, además de ser muy elevado (aproximadamente 7 segundos por cada muestra de tamaño 35 para el algoritmo de backtracking). 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

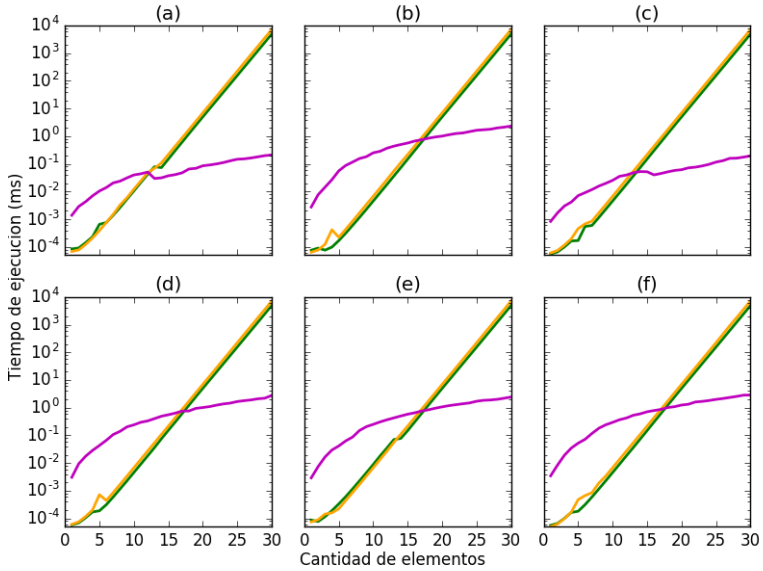

Figura 1: Tiempo de ejecución en función del tamaño de la entrada (graficados en escala logarítmica) de los algoritmos de fuerza bruta (✓) , backtracking (✓) y programación dinámica (✓) para los casos: w_i, v_i al azar entre 0 y 100 (a) y 0 y 1000 (b); w_i al azar entre 0 y 100, $v_i = w_i + 10$ (c); 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 1 y 2 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) 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 1c donde el tiempo de ejecución del algoritmo de programación dinámica disminuye levemente alrededor del punto en que comienza a desempeñar mejor que los otros.

El único algoritmo que mostró diferencias al ser corrido con diferentes W fue al de progremación dinámica, siendo más rápido cuando W era la mitad de la suma de los costos. Esto se debe 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.

Para ninguno de los tres algoritmos se vio diferencias entre el tipo de elemento que contenía el arreglo. Extrañamente, esto incluye al de backtracking, cuyo tiempo de ejecución fue incluso peor que el de fuerza bruta (en unos pocos casos y para tamaños pequeños, a lo sumo se comportó igual). Dado que el algoritmo de backtracking realiza podas, intuitivamente se esperaría que, dado la cantidad de muestras tomadas, se viera alguna mejora en la performance con respecto a generar todas las soluciones. Para ahondar en las posibles causas de esto, se intentó modificando el orden de los elementos,

tanto creciente como decreciente, sin efecto (estos resultados no se incluyeron en el informe). Esto puede deberse a que la poda por factibilidad no es suficiente para producir un efecto significativo en el tiempo de ejecución [2] o a que la mejora que aportan los casos en los que esto sucede se pierde cuando se considera una muestra grande. Es posible , también, que los casos utilizados para la experimentación no hayan permitido verlo.

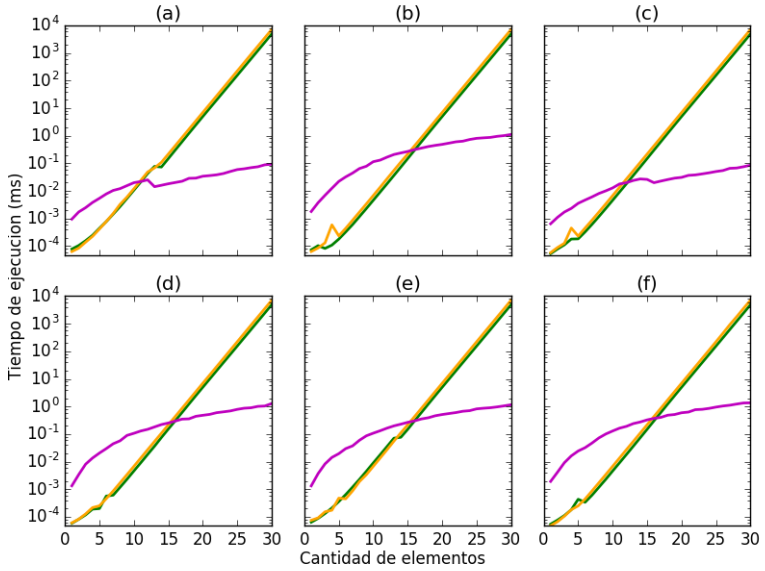

Figura 2: Tiempo de ejecución en función del tamaño de la entrada (graficados en escala logarítmica) de los algoritmos de fuerza bruta (/) , backtracking (/) y programación dinámica (/) para los casos: w_i, v_i al azar entre 0 y 100 (a) y 0 y 1000 (b); w_i al azar entre 0 y 100, $v_i = w_i + 10$ (c);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

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 teoría) un algoritmo de fuerza bruta en promedio, esto no se corresponde con los resultados obtenidos. Es posible que no baste únicamente con podar por factibilidad y sea necesaria alguna otra optimización para llegar a eso (por ejemplo, podar por optimalidad o combinar las dos).

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 por defecto que utiliza Python [2]).

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.

[2]Como se observó en el cálculo de complejidad de algoritmo de backtracking (2), la cantidad de operaciones que realiza es superior a las del de fuerza bruta (1), por más que asintóticamente se comporten igual.

Referencias

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

[2] https://en.wikipedia.org/wiki/Timsort