Universidad de Buenos Aires

Facultad de Ingeniería

75.29 Teoría de Algoritmos

Trabajo Práctico 2

Integrantes

- Arjovsky, Tomás
- Gavrilov, Seva
- Pereira, Fernando
- Pérez Dittler, Ezequiel

Segundo cuatrimestre de 2016

Índice

1. Programación dinámica

1.1. El problema de la mochila (versión 0-1)

En este problema, tenemos una cantidad n de items, cada uno de los cuales posee un peso w y un valor v, ambos no negativos. Con estos items, se requiere llenar una mochila, la cual posee una capacidad máxima determinada.

El problema plantea encontrar los items que se incluirán en la mochila, de tal forma que la suma de todos sus pesos particulares w_i no supere la capacidad máxima de la misma y, además, la suma de todos los valores particulares v_i de los objetos que se incluyen, sea máxima.

Contrario a lo que uno puede intuir, no existe un algoritmo greedy eficiente que lo resuelva, por lo que caemos en la programación dinámica como una nueva técnica para encontrar soluciones óptimas a determinados problemas, como el de la mochila, partiendo el mismo en sub-problemas cada vez más pequeños (los cuales se resolverán mucho más fácilmente), y solapando las soluciones a dichos sub-problemas para llegar a la solución del problema original. Es decir, debemos procurar resolver sub-problemas cada vez más sencillos, los cuales se utilizarán para encontrar la solución al problema mayor.

En este informe se verá un pequeño análisis general del orden de complejidad de la solución encontrada, como así también las diferencias entre los tiempos de ejecución de dos enfoques distintos para implementar la solución (Bottom-up y Top-Down).

1.1.1. Solución

En esta versión del problema de la mochila, podemos entender que un único ítem puede pertenecer a la solución óptima o no. Siendo S nuestro conjunto solución, resumiremos la misma en encontrar el valor máximo $V = \sum v_i$ con $i \in S$ que se puede incluir en la mochila sin superar su capacidad, es decir, restringido a que $\sum w_i \leq W$.

Es trivial ver que, si tenemos un caso hipotético en el que la capacidad de la mochila es $W = 100$ y el peso de un elemento $w_i = 101$, entonces, dicho elemento i queda descartado de la solución óptima. Por otro lado, si el peso del item es menor a la capacidad de la mochila, puede o no pertenecer a la solución óptima. Esto se resuelve comparando entre el valor máximo V que se puede llegar a obtener con una solución óptima *sin* el elemento corriente, y el valor máximo V que se puede obtener con una solución óptima *incluyendo* el elemento corriente. Simplemente, el mayor de estos dos valores, es la solución óptima que se está buscando.

Suponemos que tenemos n elementos $\{1 .. n\}$. Quiero encontrar la solución óptima para n elementos y una capacidad de W. Formalizando lo expresado en los últimos dos párrafos, quiero encontrar el valor máximo V_{max} que puedo obtener cumpliendo las restricciones planteadas ($V_{max} = valor\_optimo(n, W)$).

Comenzando con el elemento n, si $w_n > W \Rightarrow valor\_optimo(n, W) = valor\_optimo(n-1, W)$, ya que no puedo incluir a mi elemento n, por lo que debo encontrar una solución óptima con los $n-1$ elementos restantes de mi conjunto, y el mismo peso máximo como restricción (no se incluyó el elemento, por lo que no se ocupó espacio).

Ahora bien, si $w_n \leq W \Rightarrow valor\_optimo(n, W) = max(valor\_optimo(n-1, W), v_n + valor\_optimo(n-1, W-w_n))$. Como se planteó anteriormente, se debe encontrar el valor máximo entre una solución sin incluir al elemento corriente, y una solución incluyendo al elemento (esto es, encontrar una combinación con los n - 1 elementos restantes, ya habiéndole sumado el valor del elemento n, y habiéndole restado a la capacidad total de la mochila el peso del elemento que incluí). Esta recurrencia es la que se plantea para resolver el problema de la mochila.

1.1.2. Complejidad

Al resolver este problema por programación dinámica, uno de los puntos importantes a destacar es el de la *memoización*. Es decir, debemos utilizar alguna estructura de datos para poder guardar los resultados de cada uno de los sub-problemas, y utilizarlos cuando se los necesite nuevamente. En efecto, nos aseguramos de calcular una solución óptima para un sub-problema en particular solo una vez, y luego tomar ese resultado cuantas veces lo necesitemos en $O(1)$.

Al tomar la recurrencia planteada en la sección anterior, vemos que a nuestros sub-problemas los estamos dividiendo en base a dos parámetros:

- Cantidad de elementos
- Peso restante en la mochila

Es decir, cada cantidad de elementos distinta y peso restante distinto es un sub-problema particular a resolver. Por ende, podemos utilizar una matriz $M[n][W]$, en la cual guardaremos el valor óptimo conseguido para cada sub-problema particular (es decir, para determinada cantidad de elementos $i \in 1 \dots n$ y determinado peso de la mochila $w \in 1 \dots W$).

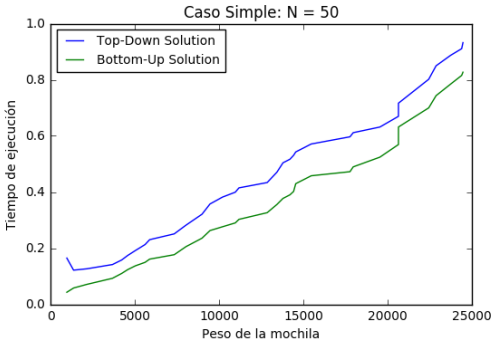
El algoritmo va resolviendo cada sub-problema particular, por lo que irá llenando la matriz planteada con el valor óptimo de cada suproblema. Si necesitamos la solución para un sub-problema ya resuelto, simplemente consultamos la matriz en O(1), por lo que el orden de nuestro algoritmo depende de la cantidad de sub-problemas a resolver, ergo, de la cantidad de elementos que posee la matriz. Por esto, podemos decir que nuestro algoritmo es $O(nW)$.

Ahora bien, es de notar que no es un orden de complejidad polinomial dependiente de n común, ya que también depende de W. Este tipo de algoritmos se los conoce como *pseudo-polinomiales*, los cuales pueden ser eficientes si los valores w_i de los ítems entrada son lo suficientemente pequeños, pero cuya complejidad aumenta muchísimo para valores muy grandes como veremos en los próximos ejemplos.

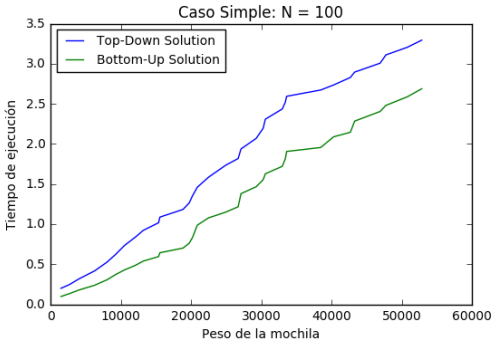
1.1.3. Tiempos de ejecución

En esta sección se encontrarán tiempos de ejecución alcanzados por las soluciones implementadas para el problema de la mochila. Vale aclarar que se implementó tanto una solución Top-Down como una solución Bottom-Up, con sus respectivas ventajas y desventajas, y justamente, uno de los enfoques de esta sección es mostrar la diferencia de tiempos entre cada una en base a las características de los parámetros de entrada (como ser el peso de la mochila o el peso de cada uno de los ítems de entrada).

Primeramente, mostramos un par de casos básicos, con 50 y 100 elementos, pesos de mochila máximo cerca de los 26000 y 52000 respectivamente, y valores y pesos de los elementos distribuidos uniformemente.

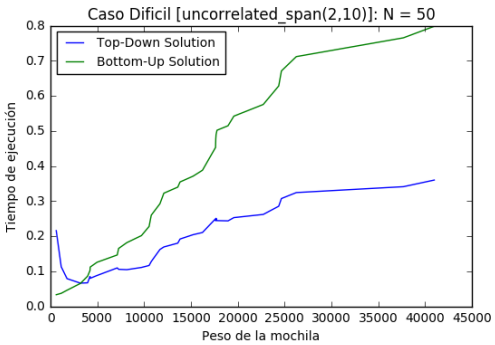


(a) $n = 50$, $W_{max} \approx 26000$

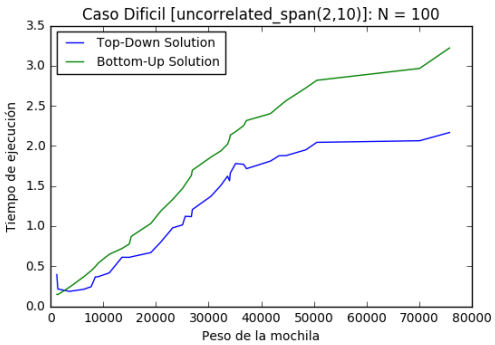


(b) $n = 100$, $W_{max} \approx 52000$

Figura 1: Instancias con pesos y valores desvinculados (uncorrelated)

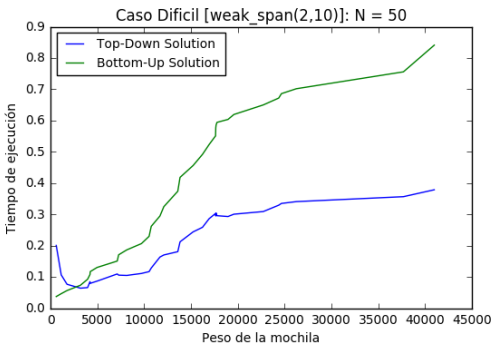


(a) $n = 50$, $W_{max} \approx 41000$

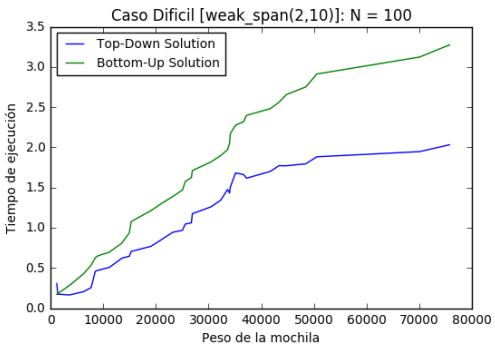


(b) $n = 100$, $W_{max} \approx 75000$

Figura 2: Instancias con pesos y valores desvinculados difícil ($uncorrelated\_span(2, 10)$)



(a) $n = 50$, $W_{max} \approx 41000$

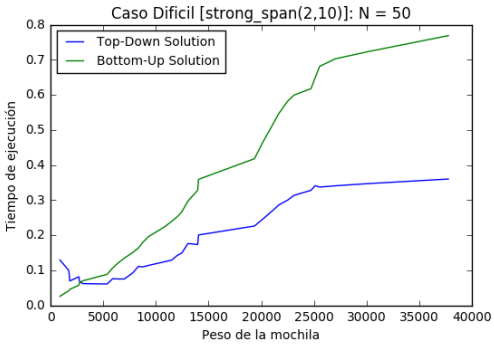


(b) $n = 100$, $W_{max} \approx 75000$

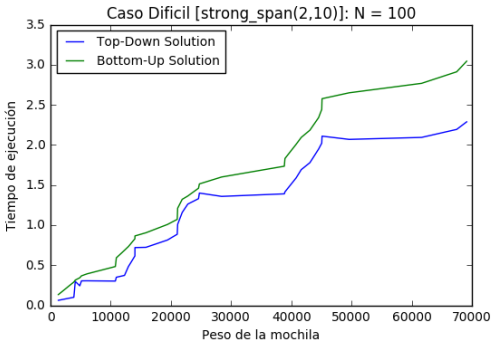
Figura 3: Instancias con pesos y valores debilmente vinculados difícil ($strongly\_correlated\_span(2, 10)$)

Como podemos ver en el caso básico inicial, los tiempos entre la solución Top-Down y Bottom-Up crecen relativamente en forma similar para esta instancia del problema, siendo la solución Bottom-Up la que mejor se ajusta. Lo que hay que notar de estos dos ejemplos es que el tiempo de ejecución del problema con $n = 100$ (para el mismo peso) es el doble del tiempo encontrado para el problema con $n = 50$. Podemos ver, por ejemplo, con un peso $W = 20000$, que en el gráfico con 50 elementos, la solución Bottom-Up tardó aproximadamente 0.5

[s] y la Top-Down 0.6 [s], mientras que para 100 elementos y mismo peso, los tiempos son aproximadamente de 1 [s] y 1.2 [s] respectivamente. Un comportamiento similar (aunque un tanto más variable, por ser instancias más difíciles), se puede ver en los tiempos de las demás ejecuciones.

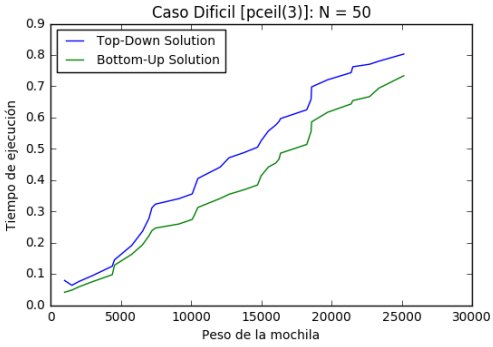


(a) $n = 50$, $W_{max} \approx 37000$

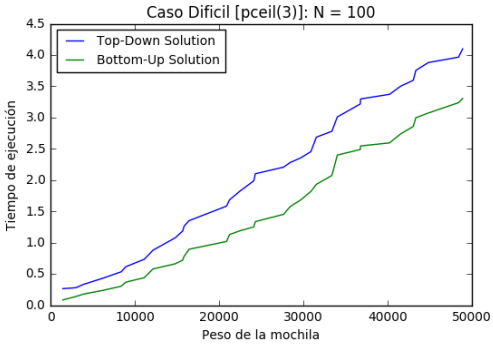


(b) $n = 100$, $W_{max} \approx 75000$

Figura 4: Instancias con pesos y valores fuertemente vinculados difícil ($strongly\_correlated\_span(2, 10)$)

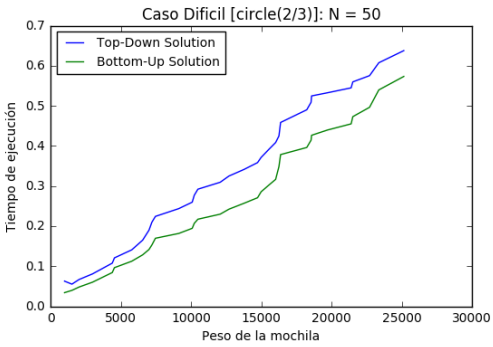


(a) $n = 50$, $W_{max} \approx 25000$

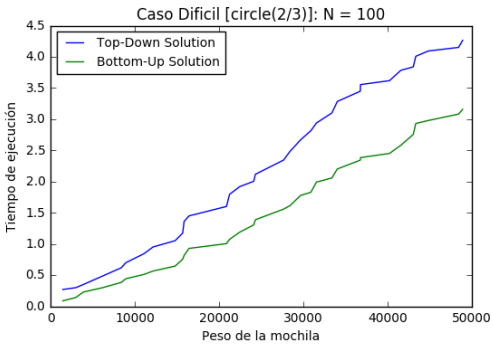


(b) $n = 100$, $W_{max} \approx 49000$

Figura 5: Instancia difícil ($pceil(3)$)



(a) $n = 50$, $W_{max} \approx 25000$



(b) $n = 100$, $W_{max} \approx 49000$

Figura 6: Instancia difícil ($circle(2/3)$)

Ahora pasamos a un caso más particular e interesante. Nos enfocamos en una instancia del problema en la

que la capacidad máxima de la mochila es grande y tenemos un set de entrada con pesos w_i similares entre sí y muy altos (tal que solo uno de los elementos entra en la mochila).

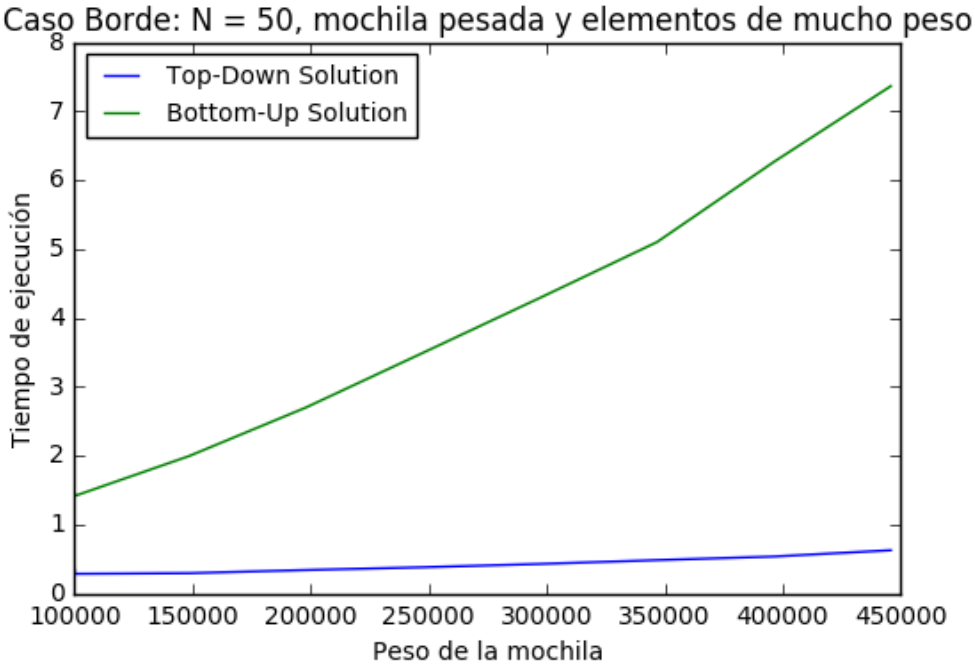


Figura 7: $n = 50$, w_i altos y similares entre sí

Una de las cosas que queríamos remarcar es la diferencia entre los tiempos de ejecución de la implementación Top-Down y la Bottom-Up en este tipo de instancias del problema.

La enorme diferencia se debe básicamente a la forma de resolver el problema que tiene cada técnica. La implementación Bottom-Up va resolviendo desde los sub-problemas más pequeños hasta llegar al problema final deseado iterativamente, obteniendo la solución óptima para absolutamente todos los sub-problemas cuyos parámetros de entrada son menores o iguales al problema original. La desventaja de esta técnica, es que está desperdiciando mucho tiempo en resolver sub-problemas que *podrían no utilizarse* para resolver el problema deseado. En otras palabras, la solución Bottom-Up llena completamente la matriz M de resultados óptimos, cuando hay muchos sub-problemas que no son necesarios.

Por otro lado, la solución Top-Down arranca desde el problema con los parámetros originales que queremos resolver, y recursivamente va partiendo el original en sub-problemas y resolviéndolos hasta obtener todos los resultados deseados. Es decir, la solución Top-Down solo se enfoca en resolver los sub-problemas estrictamente necesarios en los que se divide el problema original, sin gastar tiempo de cómputo en sub-problemas cuya solución jamás utilizaríamos. La desventaja de la solución Top-Down es el overhead que puede traer una solución recursiva, y el espacio en el stack que ésta requiere, el cual se reduce de cierta forma utilizando correctamente variables globales.

Informalmente, podemos decir que la solución Bottom-Up se toma su tiempo en resolver absolutamente todo, mientras que la solución Top-Down va al grano y resuelve lo estrictamente necesario. Es por eso que en este tipo de problemas, la solución Top-Down puede resultar mucho más eficiente que la Bottom-Up.

1.2. El problema del viajante de comercio

El problema del viajante consiste, dado n ciudades a visitar, en encontrar un camino de costo mínimo que recorra una única vez cada ciudad, regresando finalmente al origen. Dirigirse de una ciudad a otra tiene un costo, pudiendo ser simétrico o no.

El problema del viajante de comercio se hizo muy popular en la década del 50 y 60, luego de que se presentara el artículo "Solution of a large-scale traveling-salesman problem" (Dantzig, Fulkerson, and Johnson 1954) donde resuelve el problema para 49 ciudades (una por cada estado de EEUU y Washington). El método propuesto por el artículo usa técnicas de programación lineal, aún cuando siquiera existían los programas informáticos.

En el año 1962 se presentan simultáneamente y en forma independiente los artículos "A dynamic programming approach to sequencing problems" (Held and Karp 1962) y "Dynamic programming treatment of the travelling salesman problem" (Bellman 1962). Ambos proponen el uso de la programación dinámica como técnica para encontrar la solución del problema del viajante.

1.2.1. Solución

Se define la función $D(v, S)$ la distancia mínima desde v hasta la ciudad de origen, S el conjunto de ciudades a visitar. Si el conjunto S se encuentra vacío, $D(v, S) = d_{v0}$. Se define d_{ij} como la distancia desde la ciudad i hasta la ciudad j. Para el resto de los casos, $D(v, S) = \text{mín}_{u \in S}(d_{vu} + D(u, S - \{u\}))$

$$D(v, S) = \begin{cases} c_{vv0} & \text{si } S = \varnothing \\ \text{mín}_{u \in S}[c_{vu} + D(u, S - u)] & \text{otro caso} \end{cases}$$

Un pseudocódigo para calcular la distancia del ciclo hamiltoniano mínimo es el siguiente:

```
function TSP (M, n)
  for k := 2 to n do
    C({1, k}, k) := M[1,k]
  end for

  for s := 3 to n do
    for all S in {1, 2, . . . , n}, |S| = s do
      for all k in S do
        C(S, k) = min [C(S - {k}, m) + M[m,k]]
      end for
    end for
  end for

  opt := min[C({1, 2, 3, . . . , n}, k) + M[k,1]]
  return (opt)
end
```

1.2.2. Complejidad

Al ser el problema del viajante de complejidad NP-completo, no existen algoritmos que permitan tomar decisiones para encontrar la solución. Esto implica que se deben evaluar todas las soluciones posibles y elegir

el valor de menor costo.

El número de ciclos posibles del problema del viajante se puede calcular de la siguiente manera: desde el origen restan $(n-1)$ ciudades para empezar, luego se debe elegir cualquiera de las $(n-2)$ ciudades restantes y así sucesivamente. De esta forma, multiplicando todas las cantidades se obtiene el número total de caminos posibles:

$$(n-1)! = (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1$$

Por lo tanto, el método directo implica evaluar $(n-1)!$ soluciones posibles, siendo su orden $O((n-1)!)$. En el caso particular de 10 ciudades esto significaría evaluar 362880, es decir, para un problema no excesivamente grande el número de caminos posibles aumenta considerablemente.

La cantidad de operaciones fundamentales empleadas algoritmo de Held–Karp es:

$$\left(\sum_{k=2}^{n-1} k(k-1)\binom{n-1}{k}\right) + (n-1) = (n-1)(n-2)2^{n-3} + (n-1)$$

Por lo tanto, su orden temporal es $O(n^2 2^n)$. Para el caso particular de $n = 10$ el número de soluciones se reduce de 362880 a 102400.

Si se asigna una unidad de espacio al número $D(v, S)$ la cantidad de espacio requerida es:

$$\left(\sum_{k=2}^{n-1} k\binom{n-1}{k}\right) + (n-1) = (n-1)2^{n-2}$$

Por lo tanto, su orden espacial es $O(n2^n)$.

1.2.3. Tiempos de ejecución

2. Flujo de redes

Referencias

Bellman, Richard. 1962. "Dynamic Programming Treatment of the Travelling Salesman Problem." *Journal of Association for Computing Machinery.*

Dantzig, George, Delbert Fulkerson, and Selmer Johnson. 1954. "Solution of a Large-Scale Traveling Salesman Problem." *Operations Research 2.*

Held, Michael, and Richard M. Karp. 1962. "A Dynamic Programming Approach to Sequencing Problems." *Journal for the Society for Industrial and Applied Mathematics.*