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.

1.2. El problema del viajante de comercio

2. Flujo de redes