CS472 Module 7 Part C - Regular Expressions

Athens State University

February 29, 2016

Outline

Contents

1 Knapsack Problems

Recall: The 0-1 Knapsack Problem

- Given n items of known weights $w_1, w_2, \ldots, w_n$ and a knapsack of capacity W, what is the most valuable subset of the items that fit into the knapsack
- Example of an optimization problem: find the best solution given some sort of optimization function
  - In the case of the knapsack problem, combination of items whose total weight does not exceed the capacity of the sack

Example: Knapsack Problem: A (Very) Bad Algorithm

- Consider every possible subset of items
- Compute the total weight and discard if more than W
- Choose the remaining subset with maximum total value
- This is a $O(2^n)$ process!

2 Greedy Knapsacks

A Greedy Approach to the Knapsack Problem

- Let's be smart about the problem: put the items into the sack in non-increasing order according to value

– This strategy has a problem: what if the most profitable item has a large weight in comparison to its value?

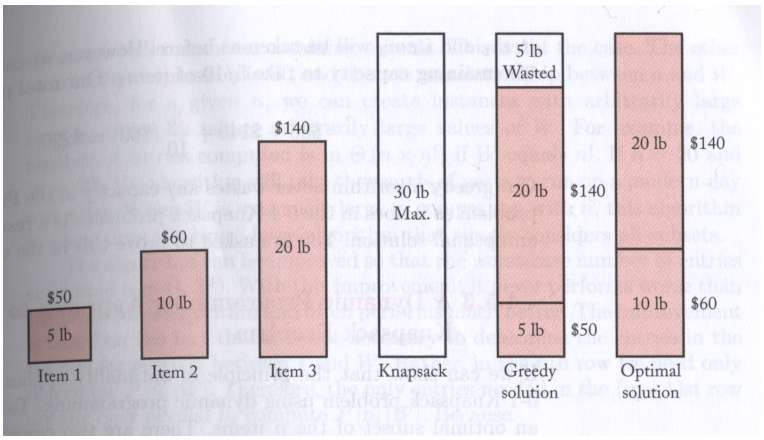
Example: Three items: (1) 25 lbs. with a value of \$10, (2) 10 lbs. with a value of \$9, and (3) 10 lbs. with a value of \$9

* If the sack can only hold 30 lbs., this strategy will yield a value of only \$10 while the optimal solution is \$18

A Different Greedy Approach to the Knapsack Problem

- Let's take this approach: normalize the value by weight; i.e., take the item with the largest value per unit weight first
- Order the items by value per unit weight and select them in sequence
- Put an item in the sack if its weight does not bring the total weight about W

A Different Greedy Approach to the Knapsack Problem



The Fractional Knapsack Problem

- What if we were to allow a person to break each item input smaller parts?
  - This is known as the *Fractional Knapsack Problem*
- Analogy: 0-1 Knapsacks are working gold ingots, Fractional Knapsacks are working with gold dust
- If permit a person to break up items, then the "sort and select by value per unit weight" greedy approach will always find the optimal solution to this problem

3 Dynamic Knapsacks

Dynamic programming and the 0-1 Knapsack problem

- Must show that the problem fulfills the principle of optimality
- Let A be a an optimal subset of the n items
- Two cases: A contains the n-th item or it does not contain the n-th item
  - If A does not contain the n-th item, then A is equal to an optimal subset of the first $n-1$ items
  - If A does contain the n-th item, then the total value of the items in A is equal to v_n, the optimal value of the $n-1$ items under the restriction that the weight cannot exceed $W - w_n$

Knapsacks - DP Style

- For $i > 0$ and $w > 0$, let $V[i][w]$ be the optimal value of items obtained from selecting from the first i items under the restriction that the weight cannot exceed w
- So, in general, we have

$$V[i][w] = \begin{cases} \max(V[i-1][w], v_v + V[i-1][w-w_i]) & w_i \leq w \\ V[i-1][w] & w_i > w \end{cases} \quad (1)$$

Knapsacks - DP Style: Implementation

- Define a 2-d array V with rows indexed from 0 to n and columns indexed from 0 to W
- Compute the rows of the array in sequence using the previous expression for $V[i][w]$
- Set the values for $V[0][w]$ and $V[i][0]$ to be 0

Knapsacks - DP Style: Analysis

- It is fairly simple to see that the number of array entries computed is equal to nW which means the algorithm is $\Theta(nW)$
- Wait... did you say linear in n? Yes, but note that the product is nW!
- So, the algorithm will perform poorly if the value of W is extremely large in comparison to n
- But we can improve things by making the observation that we are doing too much work if compute forward from 0 to n

Knapsacks - DP Style: We can do better

- Note that we don't need to determine the entries for the ith row for every w between 1 and W
  - In the nth row, we need only to determine $V[n][W]$
- So...

$$V[n][W] = \begin{cases} \max(V[n-1][W], v_n + V[n-1][W-w_n]) & w_n \leq W \\ V[n-1][W] & w_n > W \end{cases}$$

Knapsacks - DP Style: We can do better

- This means that the only entries in the $(n-1)$st row are $V[n-1][W]$ and $V[n-1][W-w_n]$
- Thus, we can work backwards with the understanding that, for the $(i-1)$st row, $V[i][w]$ is computed from $V[i-1][w]$ and $V[i-1][w-w_i]$
- The optimized algorithm stops when $n = 1$ or $w \leq 0$. This approach is $\Theta(2^n)$

4 So, what's the point?

So, what's the point

- So, we have algorithms that take two different approaches to solving the same problem
- We have a real-world problem that can be solved using one of these knapsack problems. Which algorithm to do we implement?
  - The blind greedy algorithm gives a quick approach that will not always give us an optimal solution
  - We have a DP algorithm that can be done in $\Theta(nW)$ time that can possibly be optimized to $\Theta(2^n)$.
  - A similar analysis can be used to create a $\Theta(2^n)$ divide-and-conquer algorithm that will not be bound by the problem size
- The DP algorithm is preferable in this case but it does much less re-work that the D-and-C algorithm
- But this problem is also NP-complete!