CSC 373: Algorithm Design and Analysis
Lecture 6

Allan Borodin

January 18, 2013

Lecture 6: Outline

- Start dynamic programming (DP)
- The weighted interval selection problem (WISP)
- The knapsack problem

Dynamic programming

- Dynamic programming (DP) began as and remains a very general algorithmic approach for solving optimization problems.
- Its usage now goes beyond that but still optimization is the main use.
- We start with our first problem, namely interval selection but now we consider the weighted version.

The weighted interval selection problem (WISP)

- Interval I_j starts at s_j, finishes at f_j, and has weight (or profit) w_j.
- Two jobs are compatible if they don't overlap. (Say we allow $f_i = s_j$)
- **Goal:** find a mutually compatible set of intervals S so as to maximize the sum of interval weights in the chosen set.

- Can we use a greedy algorithm?

Why not use greedy for WISP?

- All the possible ways of ordering the input items that we can think of will not only fail to be optimal but can produce arbitrarily bad solutions.
- Some possible orderings: by non increasing weight, by non increasing weight/interval length.
- Moreover, for a general greedy formalization it can be proven that no greedy algorithm can provide a good solution (in the worst case).
- There are some extensions to a greedy approach which do allow constant approximations (i.e. by allowing revocable acceptances) and even optimality (i.e. by a local ratio/primal dual algorithm that uses a reverse delete phase).

The DP approach

- Let's consider an optimal solution and once again assume that the intervals have been sorted by non-decreasing finishing time.
- Then in an optimal solution OPT, either the last interval I_n was selected or it was not.
  - If not, then we must be using an optimal solution for the first $n-1$ intervals.
  - If I_n is in OPT then no interval in OPT after time s_n.
  - Furthermore (and this is the essential aspect of DP), the intervals ending by s_n must be chosen optimally.

Note

- Once again we will define the problem so that an interval can start when another one ends.
- We can easily modify things if we do not want to allow an interval to start at precisely the time another ends.

The value/profit of an optimal solution

- The previous observation leads us to compute the entries (for $i = 1, \ldots, n$) in the following "semantic array"

$V[i] =$ max profit obtainable by a set of intervals which are a subset of the first i intervals $\{I_1, \ldots, I_i\}$

- The optimal value then is $V[n]$.
- We can also define $V[0] = 0$.
- To compute the entries of this array, it is helpful to define

$pred(i) =$ the largest index j such that $f_j \leq s_i$

(If we allowed a job to start where another ended we would then have $f_j < s_i$.)

Recursively computing the $V[i]$

- $V'[0] = 0$
- $V'[i] = \max\{A, B\}$ for $i > 1$, where

$$A = V'[i-1] \text{ and } B = V'[pred(i)] + w_i.$$

- Here B (resp. A) corresponds to the case that the ith interval is used (resp. not used) in the optimum solution for the first i intervals.
- We can arbitrarily assume that we take the solution corresponding to case A when $A = B$.

Claim

$$V[i] = V'[i] \quad \text{for all } i = 1, 2, \ldots, n.$$

Iterative vs recursive implementation

- We can clearly compute the entries of $V'[i]$ iteratively for $i = 0, 1, \ldots, n$. Time bound is $O(n \log n)$ for sorting and for computing pred[i] values.

- What if we use a recursive program directly following the definition of V'?
  - Suppose for all $i = 1, 2, \ldots, n-1$, interval I_i overlaps I_{i+1} and no other I_j for $j > i+1$.
  - This leads to the complexity recurrence

$$T[n] = T[n-1] + T[n-2]$$

    whose solution (recall Fibonacci sequences) is exponential in n.

- Memoization avoids this problem. In some sense, memoization is a defining characteristic (say verses divide and conquer) of DP algorithms.

Why two arrays V and V'?

- The semantic array is defined to say what we are trying to compute.
- The recursively defined computational array is essentially a high level code for how to compute the entries of the semantic array.
- The creative aspect of DP is coming up with an appropriate semantic array that has to provide us with enough information to obtain the desired result as well as being easy to compute.
- And although it often seems tedious, we need a proof that $V = V'$.

Computing an optimal solution and not just the optimal value

- So far we only computed the value of an optimal solution (for WISP) but we can easily adapt the DP solution to compute the solution as well.

- While there are somewhat more efficient ways to do this, the conceptually simplest thing to do is to maintain an array, say S, where $S[i]$ contains the partial solution corresponding to the value $V[i]$.

- It should be clear from the recursion defining V' how to do this.

$$S'[i] = \begin{cases} \varnothing & \text{if } i = 0 \\ S'[i-1] & \text{if } V'[i] = V'[i-1] \\ S'[pred(i)] \cup \{i\} & \text{otherwise.} \end{cases}$$

A comment on efficient implementations of DP

Dai Tri Man Le makes the following observation on implementing a DP algorithm:

> *One problem with using DP in practice is the memory issue. When the program uses too much memory, it's no longer fast. That's why sometimes one uses recursion instead of DP, although the worst cases can be terrible. Recently I was able to improve some worst case of an algorithm used in industry from 24 hours to 5 mins using memoization. I didn't even need to memorize everything, just the most recently computed results, and it's already sufficient to see the improvement. It's also interesting that when I didn't restrict the size of the look up (hash) table as much so that it can memoize more things, the algorithm became slower. So a lot of tuning was needed for the code to perform well.*

The Knapsack problem

- In the knapsack problem we are given a set of n items $I_1, \ldots, I_n$ and a size bound B where where each item $I_j = (s_j, v_j)$ with s_j being the size of the item and v_j the value.
- A feasible set is now a subset of items S such that the sum of the sizes of items in S is at most the bound B.
- Goal: Find a feasible set S that maximizes the sum of the values of items in S.

- Often one uses w_j for the weight of the item rather than s_j but I am avoiding that due to our earlier use of w_j to denote the weight or profit of an interval in the WISP.
- In general we can allow real valued parameters but in some algorithms need to restrict attention to integral parameters. But by scaling inputs this is not a significant restriction.
- This is known to be an NP hard problem but as we shall see it is only "weakly NP hard". However, it remains an NP hard problem even when $v_j = s_j$ for all j.

A first attempt

- Here is a plausible DP approach. Lets assume all sizes are integral. Suppose we consider an optimal solution and consider the last item placed in the knapsack.
- Then after placing that item in the knapsack (say having weight s), we have reduced the available space to $B - s$.
- So it seems that we need to have a semantic array

$V[b] =$ max profit/value obtainable within size bound b for $0 \leq b \leq B$.

- The recursive array

$$V'[b] = \begin{cases} 0 & \text{for } b \leq 0 \\ \max_j \{ V'[b - s(j)] + v(j) : j = 1, 2, \ldots, n \} & \text{for } b > 0 \end{cases}$$

- Does this work and if not why not?

A correct approach

- The previous approach did not work because it allows using an item more than once.
- Instead we can use

$V[i, b]$ = the maximum profit possible using only the first i items and not exceeding the bound b.

- The corresponding computational array is :

$$V'[i, b] = \begin{cases} 0 & \text{if } i = 0 \text{ or } b = 0 \\ \max\{C, D\} & \text{if } s_i \leq b \end{cases}$$

where

$$C = V'[i-1, b] \text{ and } D = V'[i-1, b-s_i] + v_i.$$