CSC373: Lecture 8

Continue the discussion of the weighted interval selection problem

Recursive algorithms and memoization vs iterative algorithms

Obtaining a solution and not just the value of the solution

Two pseudo polynomial time algorithms for the knapsack problem

Turning the pseudo polynomial time algorithm into a fully polynomial time approximation scheme (FPTAS)

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 WISP where an interval *I(j)* is now a triple *(s(j), f(j), w(j))* where *w(j)* is the weight or profit of the *j* th interval. The objective is to find a feasible (i.e. non intersecting) set of intervals S so as to maximize the sum of interval weights in the chosen set.

Why not use greedy for WIS?

- It turns out that all the ways we can think of ordering the input items will not only fail to be optimal but can produce arbitrarily bad solutions. Moreover, for a general greedy formalization it can be proven that no greedy algorithm can provide a good solution (in the worst case).
- Some possible orderings: by non increasing weight, by non decreasing weight/interval length.

The DP approach

- Lets 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 we cannot have any intervals in OPT ending at time *s(n)* or later. Furthermore (and this is the essential aspect of DP), the intervals ending before *s(n)* must be chosen optimally. (Note: once again we will define the problem so that an interval cannot start when another one ends. We can easily modify things if we do 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 want to compute the entries (for *i = 1, …n*) in the following "semantic array": *V[i]* = max profit obtainable by a set of intervals ending at or before time *f(i).* The optimal value then is *V[n]*. We also can define *V[0] = 0*.
- To compute the entries of this array, it is helpful to define pred(i) = largest index j such that f(j) < 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} where A = V'[i-1] and B = V'[pred(i)] + w(i)
- Here B (resp. A) corresponds to the case that the *i* th interval is used (resp. not used) in the optimum solution for the first *i* intervals. (We can arbitrarily assume we take the solution corresponding to case A when A = B).
- *Claim: V[i] = V'[i] for all i = 1,2,…n.*

Iterative vs recursive implementation

- We can clearly compute the entries of *V'[i]* iteratively *for i = 0,1,…,n.* Time bound is *O(n log n)* for sorting and for computing *pred[i]* values.
- What is we use a recursive program directly following the definition of V'? Suppose for all *i = 1,2,..n-1*, *interval I(i)* overlaps *I(i+1)* and no other *I(j)* for *j > i+1*. We would be led 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.

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 a essentially 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 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[i] which contains the partial solution corresponding to the value in V[i]. It should be clear from the recursion defining V' how to do this.

The Knapsack problem

- In the knapsack problem we are given a set of *n* items *I(1), …, 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. (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) which corresponded to the weight or profit of an interval in the WISP.) In general we can allow real valued parameters but in some cases need to restrict attention to integral parameters.
- A feasible set is now a subset of items *S*: the sum of the sizes of items in *S* is at most the bound *B*. The goal is to find a feasible set *S* maximizing the sum of the values of items in *S*. This is know to be an NP hard problem but as we shall see it is only "weakly" NP hard. (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 <= b <= B.
- The recursive array V'[b] = 0 for b <= 0 and then V'[b] = max {V'[b-s(j)] + v(j): j = 1,2,…n}
- 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'[0,b] = V'[i,0] = 0. V'[i,b] = max {A,B}* where *A = V'[i-1, b]* and *B = V'[i-1, b-s(i)] + v(i)* if *s(i) <= b*; else 0