Algorithms: Design and Analysis, Part II

Approximation Algorithms for NP-Complete Problems

A Dynamic Programming Heuristic for Knapsack

Arbitrarily Good Approximation

Goal: For a user-specified parameter $\epsilon > 0$ (e.g., $\epsilon = 0.01$) guarantee a $(1-\epsilon)$-approximation.

Catch: Running time will increase as ϵ decreases.
(i.e., algorithm exports a running time vs. accuracy trade-off).

[Best-case scenario for NP-complete problems]

The Approach: Rounding Item Values

High-level idea: Exactly solve a slightly incorrect, but easier, knapsack instance.

Recall: If the w_i's and W are integers, can solve the knapsack problem via dynamic programming in $O(nW)$ time.

Alternative: If v_i's are integers, can solve knapsack via dynamic programming in $O(n^2 v_{\max})$ time, where $v_{\max} = \max_i \{v_i\}$.
(See separate video)

Upshot: If all v_i's are small integers (polynomial in n) then we already know a poly-time algorithm.

Plan: Throw out lower-order bits of the v_i's!

A Dynamic Programming Heuristic

Step 1 of algorithm:

Round each v_i down to the nearest multiple of m
[larger $m \Rightarrow$ throw out more info $\Rightarrow$ less accuracy]
[Where m depends on ϵ, exact value to be determined later]

Divide the results by m to get $\hat{v}_i$'s (integers). (i.e., $\hat{v}_i = \lfloor \frac{v_i}{m} \rfloor$)

Step 2 of algorithm: Use dynamic programming to solve the knapsack instance with values $\hat{v}_1, \ldots, \hat{v}_n$, sizes $w_1, \ldots, w_n$, capacity W.

Running time $= O(n^2 \max_i \hat{v}_i)$

Note: Computes a feasible solution to the original Knapsack instance.