Approximation Algorithms for NP-Complete Problems

A Dynamic Programming Heuristic for Knapsack

Algorithms: Design and Analysis, Part II

Arbitrarily Good Approximation

Goal: for a user-specified parameter $\varepsilon > 0$ (e.g., $\varepsilon = .01$) guarantee a $(1-\varepsilon)$-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 $O(n^2 \cdot v_{max})$ 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 → throw out more info → 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., $\hat{v}_i = \left\lfloor \frac{v_i}{m} \right\rfloor$)

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

Note: Computes a feasible solution the original knapsack instance.

Tim Roughgarden