CSC 2421 - Approximation Algorithms February 10, 2005

Lecture 6: Dynamic Programming[1]

1. Solving General Knapsack Problem with Dynamic Programming

Last time we considered two dynamic programming algorithms for the general knapsack problem (GKS):

Given a set of bulk-value pairs $(b_i, v_i), 1 \leq i \leq n$ and the knapsack capacity B such that $b_i \leq B \; \forall i$, find a feasible solution $S \subseteq \{1, ..., n\}$ such that $\sum_{i \in S} b_i \leq B$ and $\sum_{i \in S} v_i$ is maximum possible

One solution based on dynamic programming (DP) is as follows. Let $M[i, v]$ be the minimum weight of a feasible solution S having profit at least v and using first i items, where $0 \leq i \leq n$ and $0 \leq v \leq \sum v_i \leq n \cdot v_{max}$. Then we can compute $M[i, v]$ using recursion:

Base cases:

$$\begin{aligned} M[i, v] &= 0 && \text{if } v \leq 0, \text{ and} \\ M[0, v] &= \infty && \text{if } v > 0. \end{aligned}$$

Recursion:

$$M[i, v] = \min \begin{cases} M[i-1, v] & \text{if } v_i \text{ is not used} \\ M[i-1, v - v_i] + b_i & \text{if } v_i \text{ is used} \end{cases}$$

The problem is NP-hard. To make it polynomial-time solvable, we break the $[0, v_{max}]$ interval into polynomial number of sub-intervals of length s and take

$$\tilde{v}_i = \left\lceil \frac{v_i}{s} \right\rceil \cdot s \quad \text{and} \quad \hat{v}_i = \left\lceil \frac{v_i}{s} \right\rceil$$

Then $s = \frac{\varepsilon}{n} \cdot v_{max}$. Assume, without loss of generality, that n/ε is an integer. Then the optimum solution $\tilde{S}$ for $\tilde{v}_i$'s is equal to the optimum solution $\hat{S}$ for $\hat{v}_i$'s. Also,

$$\hat{v}_{max} = \frac{v_{max}}{\frac{\varepsilon}{n} \cdot v_{max}} = \frac{n}{\varepsilon}.$$

Optimal solution $\hat{S}$ can be computed in $O(n \cdot n^2/\varepsilon) = O(n^3/\varepsilon)$, where n^2/ε is the number of solutions maintained for each of the n steps.

Claim: 1. *(Showing FPTAS for GKS) Let S^* be the optimal solution for values $\{v_i\}$. Then*

$$\sum_{i \in S^*} v_i \leq (1 + \varepsilon) \cdot \sum_{i \in \hat{S}} v_i.$$

For small ε, $1 + \varepsilon \approx \frac{1}{1-\varepsilon}$, which gives an equivalent inequality:

$$\sum_{i \in \hat{S}} v_i \geq (1 - \varepsilon) \cdot \sum_{i \in S^*} v_i$$

Proof:

[1] Lecturer: Alan Borodin
Note Take: Dmitry Denisenko

$$\begin{aligned}
\sum_{i\in S^*} v_i \le \sum_{i\in S^*} \tilde{v}_i & \qquad \text{since } \tilde{v}_i \ge v_i \\
\le \sum_{i\in \widetilde{S}} \tilde{v}_i & \qquad \text{since } \tilde{S} \text{ is optimal} \\
\le \sum_{i\in \widehat{S}} (v_i + s) & \qquad v_i \le \tilde{v}_i \le v_i + 1 \\
\le n\cdot s + \sum_{i\in \widehat{S}} v_i & \\
\le \varepsilon \cdot v_{max} + \sum_{i\in \widehat{S}} v_i & \\
\le (1+\varepsilon)\cdot \sum_{i\in \widehat{S}} v_i & \qquad \text{since } v_{max} \le \sum_{i\in \widehat{S}} v_i \text{ assuming } b_i \le B\ \forall i
\end{aligned}$$

□

Ibarra and Kim[1] came up with the algorithm above thus providing an FPTAS for the GKS.

We can improve the runtime of an FPTAS as follows: let $OPT(T)$ = max profit using only items $v_i \ge T$. Break interval $[T, OPT(T)]$ into length k intervals. What is the largest weight can one get with items $v_i \ge T$? *If* we know $OPT(T)$, we can define

$$\begin{aligned}
M[i,j] = &\text{ minimum weight using largest } i \text{ items with value} \ge T \\
&\text{ and profit bigger or equal to } jk.
\end{aligned}$$

Then

$$\frac{T}{k} \ge j \ge \frac{OPT(T)}{k}.$$

FPTAS algorithm:

(1) Knowing $OPT(T)$ determine scaling factors k and T.
(2) Do dynamic programming on items with value $\ge T$. As soon as we run out of them, greedily fill the knapsack with the remaining items, ordered non-increasingly by v_i/b_i.

Additive error of the algorithm above is

$$\frac{k\cdot OPT(T)}{T} + T,$$

where the first term comes from the first step of the algorithm and the second term comes from the second step. This gives us good choices for k and T:

$$k = \frac{\varepsilon^2 \cdot OPT(T)}{4} \qquad \text{and} \qquad T = \frac{\varepsilon \cdot OPT(T)}{2},$$

giving us the following upper bound on the error:

$$error \le \frac{\varepsilon \cdot OPT(T)}{2} + \frac{\varepsilon}{2} \cdot OPT(T) \le \varepsilon \cdot OPT(T).$$

2. Taking stock

The algorithms we looked at so far consider one item at a time and maintain a number of partial solutions. Priority algorithms maintain only one partial solution, whereas DP algorithms maintain more than one. Therefore, dynamic programming can be viewed as a generalization of the priority algorithm. Furthermore, if an algorithm maintains an exponential number of solutions, it will get the optimal result. Therefore, the number of solutions maintained versus the error bound achieved is a basic trade-off in algorithm design.

For example, we already saw that there doesn't exist a constant approximation priority algorithm for the simple knapsack problem. However, maintaining two solutions gives a constant approximation algorithm. The natural questions arises: will maintaining three solutions do even better? We will show that a constant $O(1/\varepsilon^2)$ number of solutions will give a constant $O(1/\varepsilon)$ approximation ratio.

3. DP algorithm for WISP$_1$

Let us go back to interval scheduling with arbitrary profits problems (WISP$_1$ and WISP$_m$) for a moment. There does exist an optimal stack algorithm for WISP$_1$ but not for WISP$_m$.

Here is an algorithm for WISP$_1$:

Given $\{(s_i, f_i, v_i)\}$ for $i \in \{1, ..., n\}$, sort the intervals by their finish times in non-decreasing order: $f_1 \le f_2 \le ... \le f_n$. Define

$$OPT[i] = \text{maximum profit ending by time } f_i \text{ and using only intervals in } \{1, ..., i\}.$$

Let us also define the preceeding interval function $\pi(i)$ as follows:

$$\pi(i) = \max_k \{f_k \le s_i\}.$$

Then

$$OPT(i) = \max \begin{cases} OPT[i-1] & \text{if } v_i \text{ is not used} \\ OPT[\pi(i)] + v_i & \text{if } v_i \text{ is used} \end{cases}$$

It can be easily generalized to an $O(n^m)$ algorithm for m machines by defining

$$OPT[i_1, ..., i_m] = \text{maximum profit for } m \text{ machines ending by time } f_{i_j} \\ \text{on machine } j, \text{ where } i_1 \le ... \le i_m.$$

The correctness of the algorithm is proven by induction on i (but the computation would normally be done iteratively). Note that the WISP$_1$ model maintains i solutions at step i.

Our grand model for these types of algorithms is called Back-Tracking (BT). The starting point is the class of algorithms that Woeginger[2] calls "simple DP algorithms," meaning that they do induction on the number of items (the common shortest path DP algorithm does not fall into the "simple DP" category because it keeps track of path lengths, not nodes, processed).

For the WISP_1 algorithm above, at any point in time we look at the next item and decide which solution we want to extend. At every step i we maintain at most i partial solutions. The final solution is the best leaf solution of this exploration.

There are two ways to order the items: do it once before any exploration starts, or reorder after every step of the exploration. This leads to two types of simple BT algorithms (similar to two types of priority algorithms):

Definition: 1. *Algorithm that orders items to be considered before exploration starts is called* Fixed Order Back-Tracking *algorithm.*

Definition: 2. *Algorithm that picks the next item to be considered based on the path taken up to that point is a* Fully Adaptive Back-Tracking *algorithm. In an* Adaptive Back-Tracking *algorithm, the next item depends on the number of items seen thus but for each partial solution being maintained we consider the same set of items.*

It is not immediatley clear if the General Knapsack Problem can benefit from the extra power of the Adaptive (or Fully Adaptive) Back-Tracking algorithm.

It is desirable to put the DP + Greedy algorithm for the GKS into the BT model. To do that, we would have to compute the optimal solution $OPT(T)$. Because this computation is impossible (in polynomial time), we will use a good estimator of $OPT(T)$. An estimator τ is "good" if

$$\frac{OPT(T)}{2} \leq \tau \leq OPT(T).$$

This gives us values of k and T:

$$k = \frac{\varepsilon^2 \cdot \tau}{4} \quad \text{and} \quad T = \frac{\varepsilon \cdot \tau}{2}.$$

The idea is to start with some rough value of τ, say v_{max}, and maintain partial solutions corresponding to values upto 2τ. As soon as a solution with value greater than 2τ is found, double the value of τ and proceed as before, discarding half of the previously accumulated solutions (leaving at most $8/\varepsilon^2$ solutions to maintain). Doubling τ will obviously lead to doubling of k and T.

4. Lower bound width computation

It is interesting to find the lower bound on the width of the solution tree an algorithm needs to maintain in order to optimally solve the simple knapsack problem. We will give the result for an adaptive BT algorithm but it will hold for the larger class of algorithms called fully adaptive BT, which is closer to to a true backtracking algorithm.

Problem (SKS): Given set of n items with bulk b_i equal to value v_i for all $i \in \{1, ..., n\}$; size of the knapsack $B = N$; find set of items $S \subseteq \{1, .., n\}$ such that $\sum_{i \in S} v_i$ is maximized.

Theorem: 1. *The width of an optimal adaptive* BT *for* SKS *is at least* $\binom{n/2}{m/2} = \Omega(2^{n/2}/\sqrt{n})$.

The proof of this theorem, which appears in [3], is equivalent to proving the following claim:

Claim: 2. *Every subset* $Q \subseteq P$ *of size* $n/4$ *is* indispensible, *meaning that*

(1) $\exists R_Q$ *with* $n/2$ *elements such that* $\sum_{i\in Q} v_i + \sum_{i\in R_Q} v_i = N$*, and*
(2) $\nexists Q' \subseteq P$ *and* $R' \subseteq R_Q$ *such that* $Q' \neq Q$ *and* $\sum_{i\in Q'} v_i + \sum_{i\in R'} v_i = N$.

These two items can be substituted by one stronger condition:
(3) $\nexists Q_1, Q_2 \subseteq P \cup R_Q$ *such that* $Q_1 \neq Q_2$ *and* $\sum_{i\in Q_1} v_i = \sum_{i\in Q_2} v_i$.

Proof: Let every b_i be in the range $[0, \frac{8}{3}\frac{N}{n}]$. Now we will describe the strategy of an adversary that presents a list of inputs to the algorithm, asks it to order them, and then removes some items from the list. After the algorithm chooses an item, the adversary repeats the process.

There are three cases to consider:
(1) Adversarial strategy for the first $n/2$ items
(2) Adversarial strategy for the next $n/2 - 2$ items
(3) Adversarial strategy for the last 2 items

Case 1: Eliminate any v such that

$$i) \sum_{i\in Q\subseteq P} v_i + v = N, \text{ and}$$

$$ii) \ \exists Q_1, Q_2 \text{ such that } Q_1 \neq Q_2 \text{ and } \sum_{i\in Q_1} v_i = \sum_{i\in Q_2} v_i + v.$$

This will eliminate at most $3^{n/2}$ elements.

Case 2: Need to get $\sum_{i\in R_Q} v_i = N - \sum_{i\in Q} v_i$. Define

$$v_{avg} = \frac{\sum_{i\in R_Q} v_i}{n/2}.$$

Then,

$$v_{avg} = \frac{N - \sum_{i\in Q} v_i}{n/2}.$$

From

$$0 \leq \sum_{i\in Q} v_i \leq \frac{n}{4} \cdot \frac{8N}{3n} \leq \frac{2}{3} N$$

we get

$$\frac{2N}{3n} \leq v_{avg} \leq \frac{2N}{n}.$$

Therefore, v_{avg} is far away from extremes which gives us many choices. Because we never want the subsets in conditions (2) and (3) to be equal, we need to eliminate 3^n items for this case (on top of $3^{n/2}$ items that need to be eliminated for Case 1). Define $U = 3^n + 3^{n/2}$ and $J = [v_{avg} - 2U, v_{avg} + 2U]$. It is easy to see that $J \subset [0, \frac{8N}{3n}]$.

As we go along, we avoid U things and aim for items within the range J. When we go above average, take the next element smaller than average. When go below the average, pick a larger element next.

For the next $n-2$ items choose items in J. Call them R' and define

$$W = \sum_Q v_i + \sum_{R'} v_i$$

We claim two things:

(1) $|W - v_{avg}(n/2-2)| \leq 2U$
(2) $W \ll n/2 \cdot v_{avg}$

Case 3: Choose v_{n-1}, v_n. such that $\sum_{Q \cup R'} v_i + v_{n-1} + v_n = N$ and the following hold:

$$v_{n-1} + \sum_{S_1 \subseteq Q \cup R'} v_i \neq \sum_{S_2 \subseteq P \cup R} v_i,$$
$$v_n + \sum_{S_1 \subseteq Q \cup R'} v_i \neq \sum_{S_2 \subseteq P \cup R} v_i,$$
$$v_{n-1} - v_n + \sum_{S_1 \subseteq Q \cup R'} v_i \neq \sum_{S_2 \subseteq P \cup R} v_i, \text{ and}$$
$$v_n - v_{n-1} + \sum_{S_1 \subseteq Q \cup R'} v_i \neq \sum_{S_2 \subseteq P \cup R} v_i.$$

Let $V = n/2 \cdot v_{avg} - W$ and take v_n, v_{n-1} in $(V/2 - i, V/2 + i)$ for some $i \leq 2U$.

If any partial solution Q is omitted, the algorithm gets the profit of at most $N-1$.

The optimal solution gets profit $N-1$ resulting in the ratio of $1 - 1/N$. $\square$

Corollary: $(1-\varepsilon)$-approximation requires width $\Omega(1/\varepsilon)^{1/3.17} = \Omega(1/\varepsilon)^{1/2\, log_2 3}$.

Claim: 3. *The proof extends with a little work to an adaptive* BT *where each path is a "one-pass with revocable acceptances" (rather than an irrevocable decision) algorithm.*

Claim: 4. *To get optimal solution,* WISP_1 *requires* $O(n)$ *width and* WISP_m *requires* $\Omega(\binom{n}{m})$ *width for any fixed* m.

Theorem: 2. *For fixed order* BT *with revocable acceptances, width* $\gamma \leq \binom{\frac{n-1}{2}}{m}$, *the approximation ratio for* WISP_m *with proportional profits is at most*

$$1 + \delta - \frac{1}{2m(2\gamma^{1/m+1})}.$$

For example, WISP_1 has $1 - \frac{1}{4\gamma+2}$. If $\gamma = 1$ then get $5/6 = 0.833$ *but* we do not know how to realize that. Horn has $1/1.17 = 0.85$ lower bound for adaptive BT.

References

[1] O.H. Ibarra and C.E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22:463-468, 1975.

[2] G. Woeginger. When does a dynamic programming formulation guarantee the existence of a fully polynomial time approximation scheme (FPTAS)? INFORMS *Journal on Computing*, 12:57–75, 2000.

[3] M. Alekhnovich, A. Borodin, J. Buresh-Oppenheim, R. Impagliazzo, A. Magen, T. Pitassi. Toward a model for backtracking and dynamic programming.