

The Knapsack problem

- The optimization problem KNAPSACK-OPT is:

Instance: $(w_1, p_1), \dots, (w_n, p_n), W$, where $w_i, p_i, W \in \mathbb{N}$ for all $1 \leq i \leq n$

Solution: $I \subseteq \{1, \dots, n\}$ such that $\sum_{i \in I} w_i \leq W$

Objective: Maximize $\sum_{i \in I} p_i$

- Intuitively, given n objects, each having a weight w_i and a profit p_i , and given a knapsack with capacity W , the objective is to find a subset of the objects with total weight at most W , with maximum total profit.
- **Notation:** For $I \subseteq \{1, \dots, n\}$, define $w(I) = \sum_{i \in I} w_i$, and $p(I) = \sum_{i \in I} p_i$.
- KNAPSACK-OPT is **NP**-hard. By definition, this means that the corresponding decision problem KNAPSACK is **NP**-hard. The KNAPSACK decision problem is:

Instance: $(w_1, p_1), \dots, (w_n, p_n), W, P$

Question: Does there exist $I \subseteq \{1, \dots, n\}$ such that $w(I) \leq W$ and $p(I) \geq P$?

- KNAPSACK is a generalization of SUBSET-SUM. To reduce SUBSET-SUM \leq_p KNAPSACK, the idea is to define $f(\langle s_1, \dots, s_n, t \rangle) = \langle (s_1, s_1), \dots, (s_n, s_n), t, t \rangle$, i.e. map the instance $\langle s_1, \dots, s_n, t \rangle$ to an instance of KNAPSACK where $w_i = p_i = s_i$ and $W = P = t$.

A dynamic-programming algorithm

- Let $(w_1, p_1), \dots, (w_n, p_n), W$ be an instance of KNAPSACK-OPT.
- Define $A[i, j]$ to be the minimum-weight subset of $\{1, \dots, i\}$ that has profit at least j (if no such subset exists, define $A[i, j] = \infty$).
- If we knew $A[i, j]$ for “all” values of i, j then we could solve the instance of KNAPSACK-OPT by searching for the largest value of j such that $A[n, j] \neq \infty$ and $w(A[n, j]) \leq W$ and returning $A[n, j]$.
- In fact, if $P = \max\{p_i\}$ is the maximum profit, then we only need to compute $A[i, j]$ for $0 \leq j \leq n \cdot P$, since there is no subset whose profit is larger than $n \cdot P$.
- Here is a recurrence for computing $A[i, j]$ for $i \geq 1$ and $j \geq 1$:

$$A[i, j] = \begin{cases} \infty, & \text{if } A[i-1, j] = \infty \text{ and } A[i-1, j-p_i] = \infty \\ A[i-1, j], & \text{if } w(A[i-1, j]) \leq w(A[i-1, j-p_i]) + w_i \\ A[i-1, j-p_i] \cup \{i\}, & \text{otherwise} \end{cases}$$

- The base cases are:

$$\begin{aligned} A[i, j] &= \emptyset, & \text{if } j \leq 0 \\ A[0, j] &= \infty, & \text{if } j > 0 \end{aligned}$$

- The following algorithm then computes an optimal solution to the KNAPSACK-OPT instance:

```

1: for  $i \leftarrow 0, \dots, n$  do
2:   for  $j \leftarrow 0, \dots, nP$  do
3:     compute  $A[i, j]$  using the recurrence
4:   end for
5: end for
6: for  $j \leftarrow nP, \dots, 0$  do
7:   if  $A[i, j] \neq \infty$  and  $w(A[i, j]) \leq W$  then
8:     return  $A[i, j]$ 
9:   end if
10: end for

```

- If we assume that we have random access to the array, then the running time is $O(n^2P)$. Note that this is not polynomial in the input size, since the largest profit P is represented with only $O(\log P)$ input bits. (Exercise: find an instance where this algorithm takes exponential time in the input size).
- This does show that KNAPSACK is weakly **NP**-complete: if all profits are represented in unary then $O(n^2P)$ is polynomial in the input size.

Scaling

- An algorithm like the one above is often called "pseudo-polynomial time", i.e. the running time is polynomial in the input size and in the *values* of numbers occurring in the input.
- Scaling is a general technique that often leads to good approximation algorithms for problems with pseudo-polynomial time algorithms.
- The basic idea is to divide the relevant numbers (in this case the profits) by some factor, rounding if necessary, and then solve the modified instance. Note that reducing the profits actually improves the running time.
- Notice that if all of the profits have a common factor f , and we divide each profit by f , then an optimal solution for the modified instance is also an optimal solution for the original instance (since all solutions differ in value by a factor of exactly f). If f is not a common factor of all the profits, then this is no longer true, but the difference is only due to rounding, so

we can expect that the solution for the modified instance will be “close” to optimal for the original instance.

- Let I be an instance $(w_1, p_1), \dots, (w_n, p_n), W$ of KNAPSACK-OPT. Let $P = \max\{p_i\}$ be the largest profit, as before. Let K be a number (the scaling parameter) whose value we will decide on later.
- Define a new instance I' with $w'_i = w_i$, $W' = W$, and $p'_i = \lfloor p_i/K \rfloor$
- Notice that $P' = \max\{p'_i\} \leq P/K$, so we can find an optimal solution OPT' for I' in time $O(n^2 P/K)$.
- Let OPT be an optimal solution for I . Notice that, since $w'_i = w_i$ and $W' = W$, OPT is a feasible solution for I' , and thus the optimality of OPT' implies $\sum_{i \in \text{OPT}'} p'_i \geq \sum_{i \in \text{OPT}} p'_i$. So,

$$\begin{aligned}
 p(\text{OPT}') &= \sum_{i \in \text{OPT}'} p_i \\
 &= K \cdot \sum_{i \in \text{OPT}'} \frac{p_i}{K} \\
 &\geq K \cdot \sum_{i \in \text{OPT}'} \left\lfloor \frac{p_i}{K} \right\rfloor \\
 &= K \cdot \sum_{i \in \text{OPT}'} p'_i \\
 &\geq K \cdot \sum_{j \in \text{OPT}} p'_j \\
 &= K \cdot \sum_{j \in \text{OPT}} \left\lfloor \frac{p_j}{K} \right\rfloor \\
 &\geq K \cdot \sum_{j \in \text{OPT}} \left(\frac{p_j}{K} - 1 \right) \\
 &= K \cdot \sum_{j \in \text{OPT}} \frac{p_j}{K} - K|\text{OPT}| \\
 &= \sum_{j \in \text{OPT}} p_j - K|\text{OPT}| \\
 &= p(\text{OPT}) - K|\text{OPT}| \\
 &\geq p(\text{OPT}) - Kn
 \end{aligned}$$

Assume that $w_i \leq W$ for all i (if this is not the case, we can “throw away” the items that are too large, since they cannot be part of any feasible solution). This implies that $P \leq p(\text{OPT})$, since the solution containing only the most profitable item is feasible. Let $\varepsilon > 0$, and choose $K = \varepsilon P/n$.

Then we have

$$\begin{aligned} p(\text{OPT}') &\geq p(\text{OPT}) - \varepsilon \cdot P \\ &\geq p(\text{OPT}) - \varepsilon \cdot p(\text{OPT}) \\ &= (1 - \varepsilon) \cdot p(\text{OPT}) \end{aligned}$$

And the time required to compute OPT' is $O(n^2P/K) = O(n^3/\varepsilon)$.

- What have we just shown? For any, arbitrary value $\varepsilon > 0$ we can get a $(1 - \varepsilon)$ -approximate solution, in time $O(n^3/\varepsilon)$. For instance, to get a solution whose profit is at least $3/4$ of the optimal profit we choose $\varepsilon = 1/4$ and get a solution in time $O(4n^3)$ (abusing notation slightly). To a solution whose profit is at least $99/100$ of the optimal profit we choose $\varepsilon = 1/100$ and get a solution in time $O(100n^3)$. There is a cost to obtaining additional accuracy, but the cost does not grow too quickly.
- Such an algorithm is called a “Fully Polynomial-Time Approximation Scheme”, or FPTAS for short.

Approximation Schemes

- **Definition:** An FPTAS (Fully Polynomial-Time Approximation Scheme) for a maximization [resp. minimization] problem Π is an algorithm which, on input $I \in I_\Pi$ and $\varepsilon > 0$, returns a $(1 - \varepsilon)$ -approximate [resp. $(1 + \varepsilon)$ -approximate] solution for I in time polynomial in $|I|$ and $1/\varepsilon$.
- **Definition:** A PTAS (Polynomial-Time Approximation Scheme) is like an FPTAS, except that the running time is not required to be polynomial in $1/\varepsilon$ (but must still be polynomial in $|I|$).
- The algorithm we described is a proof of the following theorem.
- **Theorem:** There is an FPTAS for KNAPSACK-OPT.