Knapsack Problem

KNAPSACK PROBLEM
Given n items, $j = 1, 2, ..., n$, each with a given value c_j and size a_j, find a subset of the items which fits a given knapsack of size b and has the largest total value.

Denoting by $V = \{1, 2, ..., n\}$ and associating a binary decision variable x_j to each item $j \in V$ such that

$$x_j = \begin{cases} 1 & \text{if we put item } j \text{ into the knapsack, and} \\ 0 & \text{otherwise,} \end{cases}$$

we can formulate the above problem as the following linear binary optimization problem:

$$\begin{array}{rrcl} & \sum_{j=1}^{n} c_j x_j & \longrightarrow & \max \\ \text{s.t.} & \sum_{j=1}^{n} a_j x_j & \leq & b, \\ & x_j & \in & \{0,1\} \quad \text{for all } j \in V. \end{array} \tag{1}$$

This problem is known to be NP-hard, but as we shall see, it is among the "easiest" NP-hard problems, in some sense.

In many situation, multiple items are available of the same type, i.e., the variables are not binary, but can take arbitrary integer values within a given range.

Lemma 1 *Bounded linear integer programming is polynomially equivalent with binary linear programming.*

Proof. See in class ... □

In what follows we shall concentrate on binary knapsack problems, and assume that all input parameters are integers.

Lemma 2 *Given a knapsack problem (1), $a, c \in \mathbb{Z}^n$, and $b \in \mathbb{Z}_+$, we can assume without any loss of generality that*

$$a_j > 0, \quad c_j > 0 \quad \textit{and} \quad a_j \leq b \quad \textit{for all} \quad j = 1, ..., n. \tag{2}$$

Proof. See in class ... □

Linear programming relaxation

We shall assume in the sequel that the input consists of integer parameters and conditions (2) hold.

The LP *relaxation* of (1) can be stated as follows:

$$\begin{array}{lrcll} & \sum_{j=1}^{n} c_j x_j & \longrightarrow & \max & \\ \text{s.t.} & \sum_{j=1}^{n} a_j x_j & \leq & b, & \\ & x_j & \leq & 1 & \text{for all } j \in V, \text{ and} \\ & x_j & \geq & 0 & \text{for all } j \in V. \end{array} \tag{3}$$

This *fractional knapsack* problem is easy to solve, even easier than linear programming problems, in general.

Lemma 3 (Dantzig (1957)) *Sort the items such that*

$$\frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \cdots \geq \frac{c_n}{a_n}$$

and let k be the smallest index for which $a_1 + a_2 + \cdots + a_k > b$. Then, an optimal solution $\mathbf{x}^*$ of (3) is defined by

$$x_j^* = \begin{cases} 1 & \text{for } j < k, \\ \frac{b - \sum_{j=1}^{k-1} a_j}{a_k} & \text{for } j = k, \text{ and} \\ 0 & \text{for } j > k. \end{cases}$$

Proof. Let x be an optimal solution to (3), and let $i \neq j$ be two indices for which

$$\frac{c_j}{a_j} < \frac{c_i}{a_i}.$$

We claim that either $x_j = 0$ or $x_i = 1$. This will prove the statement (WHY?? HOW??).

To see the claim, let us assume indirectly that $x_j > 0$ and $x_i < 1$, let $\epsilon > 0$ be small enough, and define $\tilde{x}$ by

$$\tilde{x}_k = \begin{cases} x_k & \text{if } k \neq i, j \\ x_j - \epsilon a_i & \text{if } k = j \\ x_i + \epsilon a_j & \text{if } k = i \end{cases}$$

Then we have

$$\sum_{k=1}^{n} a_k \tilde{x}_k \;=\; \sum_{k=1}^{n} a_k x_k - a_j a_i \epsilon + a_i a_j \epsilon \;=\; \sum_{k=1}^{n} a_k x_k$$

and thus $\tilde{x}$ is also a feasible solution to (3) (WHY??). Furthermore, we have

$$\sum_{k=1}^{n} c_k \tilde{x}_k \;=\; \sum_{k=1}^{n} c_k x_k - c_j a_i \epsilon + c_i a_j \epsilon \;>\; \sum_{k=1}^{n} c_k x_k$$

and hence $\tilde{x}$ would be a better solution than x, contradicting the optimal choice of x. □

Let us denote by Z^{LP} the optimum value of the fractional knapsack problem (3), and let Z^{OPT} denote the binary optimum to (1). In general, if $\mathbf{A}$ is any algorithm which provides a feasible solution to the knapsack problem (1), let us denote by $\mathbf{x}^{\mathbf{A}}$ the binary vector generated by this algorithm, and let $Z^{\mathbf{A}} = \sum_{j=1}^{n} c_j x_j^{\mathbf{A}}$ denote the corresponding objective function value.

We shall say that an algorithm $\mathbf{A}$ is a ρ-factor approximation algorithm for the knapsack problem, if

$$Z^{OPT} \leq \rho Z^{\mathbf{A}}.$$

Corollary 1 *We can solve the fractional knapsack problem in $O(n \log n)$ time, and we have*

$$Z^{OPT} \leq Z^{LP} \leq 2Z^{OPT}.$$

Furthermore, choosing the better of the two sets $\{1, ..., k-1\}$ and $\{k\}$, where k is the index obtained in Lemma 3, constitutes a 2-factor approximation algorithm for the binary knapsack problem (1), which runs in $O(n \log n)$ time.

Proof. Let x^* be the optimal solution of (3), let $x^1 \in \{0,1\}^n$ be the defined by $x_j^1 = 1$ iff $j < k$, and let $x^2 \in \{0,1\}^n$ be defined by $x_j^2 = 1$ iff $j = k$. Then both x^1 and and x^2 are feasible solutions to (1), furthermore we have $x^* = x^1 + x_k^* x^2$. Thus, denoting by $C^1 = \sum_{j=1}^{n} c_j x_j^1$ and $C^2 = \sum_{j=1}^{n} c_j x_j^2$, we have

$$Z^{OPT} \leq Z^{LP} = C^1 + x_j^* C^2 \leq C^1 + C^2 \leq 2\max\{C^1, C^2\} \leq 2Z^{OPT}.$$

□

In fact, the complexity in the above statement can be reduced to $O(n)$, by using a linear time weighted median algorithm, instead of sorting (see e.g. Blum, Floyd, Pratt, Rivest and Tarjan (1973)).

MEDIAN BASED SOLUTION OF THE FKP

Initialize: Compute $q_j = \frac{c_j}{a_j}$ for $j = 1, ..., n$, and set $Z = O = \emptyset$ and $F = \{1, 2, ..., n\}$.

Main Loop: Compute $k \in F$ such that q_k is a median of $\{q_j \mid j \in F\}$, set $J = \{j \mid q_j < q_k\}$ and compute $A = \sum_{j \in J} a_j$.

(a) If $A \leq b$, then set $O = O \cup J$, $F = F \setminus J$, and $b = b - A$.

(b) If $A > b$, then set $Z = Z \cup (F \setminus J)$, $F = F \setminus (F \setminus J)$.

(c) If $|F| \geq 2$, RETURN to Main Loop.

Output: If $F = \{k\}$, then set $x_k^* = \frac{b}{a_k}$. Set $x_j^* = 1$ for all $j \in O$, $x_j^* = 0$ for all $j \in Z$, and output x^*.

Lemma 4 *If the values q_j, $j = 1, ..., n$ are all different, then the above algorithm finds a solution to the fractional knapsack problem (3) in $O(n)$ time.*

Proof. According to Lemma 3, the variable fixations in steps (a) and (b) of the **Main Loop** agree with the optimal assignment. Since all q_j values are different, about half of the variables are fixed in this way, in each main iteration. Since all operations in the preamble of the **Main Loop** can be done in at most Kn time (for some constant K), we have the following recursion for the running time $g(n)$ of the above algorithm

$$g(n) \leq Kn + g(\frac{n}{2}),$$

from which $g(n) \leq 2Kn$ follows. □

[WHAT CAN YOU DO, IF SOME OF THE q_j VALUES COINCIDE???]

Exact solution by dynamic programming

For every index $i = 1, ..., n$ and every binary vector $(x_1, ..., x_i) \in \{0,1\}^i$ let us associate two parameters, defined as

$$a(x_1, ..., x_i) = \sum_{j=1}^{i} a_j x_j \quad \text{and} \quad c(x_1, ..., x_i) = \sum_{j=1}^{i} c_j x_j.$$

Observation: If $\mathbf{x}^*$ is an optimal solution to the binary knapsack problem (1), then for every index $i = 1, ..., n$ and every binary vector $\mathbf{x} \in \{0,1\}^n$ we have

$$\text{either} \quad a(x_1, ..., x_i) \geq a(x_1^*, ..., x_i^*) \quad \text{or} \quad c(x_1, ..., x_i) \leq c(x_1^*, ..., x_i^*).$$

In other words, it is not possible to pack more value into less space out of the first i items, than in the optimal solution $\mathbf{x}^*$. Or, again in other words, the selection of items out of the first i item also has to be optimal, in this sense. [PROVE THIS OBSERVATION!]

This observation is the motivating factor behind *dynamic programming* algorithms (see Bellman (1956-57)).

DYNAMIC PROGRAMMING ALGORITHM

Initialize: Set $\mathcal{P}_1 = \{(0), (1)\}$, $k = 1$, and set $C = \lfloor Z^{LP} \rfloor \leq \sum_{j=1}^{n} c_j$.

Main Loop: For $k = 1, 2, ..., n-1$ **do**

(a) Set $\mathcal{P}_{k+1} = \emptyset$ and $\mathcal{Q}_i = \emptyset$ for $i = 1, ..., C$.

(b) For each vector $(x_1, ..., x_k) \in \mathcal{P}_k$ and for each binary value $x_{k+1} \in \{0,1\}$ compute $i = c(x_1, ..., x_{k+1})$, and set $\mathcal{Q}_i = \mathcal{Q}_i \cup \{(x_1, ..., x_{k+1})\}$, if $a(x_1, ..., x_{k+1}) \leq b$.

(c) For each $i = 1, ..., C$ for which $\mathcal{Q}_i \neq \emptyset$ choose the vector $(x_1, ..., x_{k+1}) \in \mathcal{Q}_i$ which has the smallest $a(x_1, ..., x_{k+1})$ value, and set $\mathcal{P}_{k+1} = \mathcal{P}_{k+1} \cup \{(x_1, ..., x_{k+1})\}$.

Output: Output the vector from $\mathcal{P}_n$, which has the highest $c(x_1, ..., x_n)$ value.

This algorithm solves correctly the binary knapsack problem (1), and can be implemented to run in $O(nC)$ time (WHY?? HOW??).

Such an algorithm we call *pseudo-polynomial*, because it runs in polynomial time in the unary representation of the input (but it is not polynomial the usual binary encoded input size! WHY NOT??)

Another typical presentation of this method is to define a function

$$g(k,w) = \max\left\{ \sum_{j=1}^{k} c_j x_j \;\middle|\; \sum_{j=1}^{k} a_j x_k \leq w, \quad x_j \in \{0,1\}, j = 1,...,k \right\}$$

for $k = 1,...,n$ and $0 \leq w \leq b$, integer, and realize that

$$g(k+1,w) = \max\{c_{k+1} + g(k, w - a_{k+1}), g(k,n)\}$$

corresponding to the two possible cases of $x_{k+1} = 1$ or $x_{k+1} = 0$ in the best solution of the subproblem. Clearly,

$$g(0,w) = \begin{cases} 0 & \text{if } w \geq 0, \\ -\infty & \text{if } w < 0. \end{cases}$$

Thus, the $n \times b$ "matrix" $g(k,w)$ for $k = 1,...,n$ and $w = 1,...,b$ can be filled, recursively, in $O(nb)$ time, and $g(n,b)$ will be the optimal value of (1). [WHY?? HOW CAN YOU GET AN OPTIMAL BINARY VECTOR?]

Approximation schemes

An *approximation scheme* for the knapsack problem is a family of algorithms $\mathbf{A}_\epsilon$, $\epsilon > 0$, where A_ϵ provides a $(1+\epsilon)$-factor approximation of the knapsack problem.

Such a scheme is called a *polynomial time approximation scheme* (or PTAS) if algorithm $\mathbf{A}_\epsilon$ runs in polynomial time in the input size, for every fixed value of $\epsilon > 0$. It is called a *fully polynomial time approximation scheme* (or FPTAS) if the running time of A_ϵ is bounded by a polynomial in both the input size and in $\frac{1}{\epsilon}$, for every $\epsilon > 0$.

The above dynamic programming algorithm can easily be turned into an FPTAS for the knapsack algorithm, as follows:

ALGORITHM $\mathbf{A}_\epsilon$

Initialize: Set $\mathcal{P}_1 = \{(0), (1)\}$, $k = 1$, and set $C = \lfloor Z^{LP} \rfloor \leq \sum_{j=1}^{n} c_j$. Furthermore, set $K = \lfloor \frac{\epsilon C}{2n} \rfloor$, and let $L = \lceil \frac{C}{K} \rceil \approx \frac{2n}{\epsilon}$.

Main Loop: For $k = 1, 2, ..., n-1$ **do**

(a) Set $\mathcal{P}_{k+1} = \emptyset$ and $\mathcal{Q}_i = \emptyset$ for $i = 1, ..., L$.

(b) For each vector $(x_1, ..., x_k) \in \mathcal{P}_k$ and for each binary value $x_{k+1} \in \{0, 1\}$ compute $z = c(x_1, ..., x_{k+1})$, and set $\mathcal{Q}_i = \mathcal{Q}_i \cup \{(x_1, ..., x_{k+1})\}$ if $iK \leq z < (i+1)K$ and $a(x_1, ..., x_{k+1}) \leq b$.

(c) For each $i = 1, ..., L$ for which $\mathcal{Q}_i \neq \emptyset$ choose the vector $(x_1, ..., x_{k+1}) \in \mathcal{Q}_i$ which has the smallest $a(x_1, ..., x_{k+1})$ value, and set $\mathcal{P}_{k+1} = \mathcal{P}_{k+1} \cup \{(x_1, ..., x_{k+1})\}$.

Output: Output the vector from $\mathcal{P}_n$, which has the highest $c(x_1, ..., x_n)$ value.

In this procedure we have $|\mathcal{P}_k| \leq L \approx \frac{2n}{\epsilon}$, for each $k = 1, ..., n-1$, and thus the algorithm can be implemented to run in $O(\frac{n^2}{\epsilon})$ time. At each time, choosing only one element out of $\mathcal{Q}_i$, which contains vectors with objective function value within a range of K, we may make an error of size K. Repeating this n times the total error is not more than

$$nK \leq \frac{\epsilon C}{2} \leq \epsilon Z^{OPT}.$$

These prove that the family $\mathbf{A}_\epsilon$, as defined above, form an FPTAS for the knapsack problem (WHY AND HOW, PRECISELY???).

For these results and their variations, see Ibara and Kim (1975), Sahni (1976), and Gens and Levner (1979).

Another variant can be presented as follows:

ALGORITHM $\mathbf{B}_\epsilon$

Initialize: Set $\mathcal{P}_1 = \{(0), (1)\}$, $k = 1$, and set $C = \lfloor Z^{LP} \rfloor \leq \sum_{j=1}^{n} c_j$. Furthermore, set $K = \lceil \frac{2n \ln C}{\epsilon} \rceil$, $L = \lceil \frac{2nb}{\epsilon} \rceil$, and $\mu = 1 + \frac{\epsilon}{2n}$.

Main Loop: For $k = 1, 2, ..., n-1$ **do**

(a) Set $\mathcal{P}_{k+1} = \emptyset$ and $\mathcal{Q}_{ij} = \emptyset$ for $i = 0, ..., L$ and $0 = 1, ..., K$.

(b) For each vector $(x_1, ..., x_k) \in \mathcal{P}_k$ and for each binary value $x_{k+1} \in \{0, 1\}$ compute $\gamma = c(x_1, ..., x_{k+1})$ and $\alpha = a(x_1, .., x_{k+1})$, and set $\mathcal{Q}_{ij} = \mathcal{Q}_{ij} \cup \{(x_1, ..., x_{k+1})\}$ if

$$\mu^i \leq \gamma < \mu^{i+1} \quad \text{and} \quad \mu^j \leq \alpha < \mu^{j+1}$$

(c) For each pair of indices $i = 0, ..., L$ and $j = 0, ..., K$ for which $\mathcal{Q}_{ij} \neq \emptyset$ choose the vector $(x_1, ..., x_{k+1}) \in \mathcal{Q}_i$ which has the smallest $a(x_1, ..., x_{k+1})$ value, and set $\mathcal{P}_{k+1} = \mathcal{P}_{k+1} \cup \{(x_1, ..., x_{k+1})\}$.

Output: Output the vector from $\mathcal{P}_n$, which has the highest $c(x_1, ..., x_n)$ value.

Here we have $|\mathcal{P}_k| \leq KL \approx \frac{4n^2 \ln b \ln C}{\epsilon^2}$, and thus the algorithm can be implemented to run in $O(\frac{n^3}{\epsilon^2} \ln b \ln C)$ time. Moreover, each time we choose only one vector from $\mathcal{Q}_{ij}$ we may incur a factor of $\mu = 1 + \frac{\epsilon}{2n}$ error. Since we repeat this at most n times, the total error is limited by the factor

$$\left(1 + \frac{\epsilon}{2n}\right)^n \leq e^{\frac{\epsilon}{2}} \leq 1 + \epsilon,$$

proving that indeed, the algorithms $\mathbf{B}_\epsilon$ also form an FPTAS for the knapsack problem. (AGAIN, DETAILS!! WHY AND HOW???)

For an entertaining survey about approximation schemes, see Schurman and Woeginger (2001).

Cutting-Stock Problem

Cutting-Stock Problem
A factory has to cut long rods into shorter pieces. There are b_i copies of rods of length ℓ_i ordered, for $i = 1, ..., m$. What is the minimum number of rods of length L to satisfy all orders?

A problem of this type was first considered by Eisemann (1957), and later by Gilmore and Gomory (1960, 1961). By today several generalizations of this problem (cutting 2-dimensional sheets, cutting glass, cutting textiles, etc.) were considered in the literature, many-many software packages are available to solve such problems in relatively large sizes, and there is an international cutting-stock society with over 1000 members (mostly operations researchers and computer scientists).

We consider here the simplest 1-dimensional version, as presented above, and follow the model and solution by Gilmore and Gomory (1961).

Let us call an integer vector $c = (c_1, ..., c_m) \in \mathbb{Z}_+^m$ a *cutting pattern* if

$$\sum_{i=1}^{m} c_i \ell_i \leq L,$$

or in other words, when it is possible to cut c_1 pieces of length ℓ_1, c_2 pieces of length ℓ_2, ..., and c_m pieces of length ℓ_m from one rod of length L. Let us denote by $\mathcal{C}$ the set of all cutting patterns.

Let us further introduce integer decision variables, x_c for all $c \in \mathcal{C}$. Variable x_c denotes the number of rods of length L which we plan to cut by cutting pattern c, for $c \in \mathcal{C}$. Then the cutting-stock problem can be stated as

$$\begin{array}{lrll} Z_{IP} = \min & \sum_{c \in \mathcal{C}} x_c & & \\ s.t. & \sum_{c \in \mathcal{C}} c_i x_c & \geq b_i & \text{for } i = 1, ..., m, \\ & x_c & \in \mathbb{Z}_+ & \text{for all } c \in \mathcal{C}. \end{array} \tag{4}$$

Let us note that the number of cutting patterns $|\mathcal{C}|$ is typically very large in terms of the input size. Of course, in practice most decision variables are

zero, and we can expect a good solution to be described by the few non-zero components. As an integer programming problem this is still very difficult (e.g., it generalizes set covering problems – WHY??)

One common approach is to consider the continuous relaxation

$$\begin{array}{lrll} Z_{LP} = \min & \sum_{c\in\mathcal{C}} x_c & & \\ s.t. & \sum_{c\in\mathcal{C}} c_i x_c & \geq b_i & \text{for } i = 1, ..., m, \\ & x_c & \geq 0 & \text{for all } c \in \mathcal{C}, \end{array} \tag{5}$$

and solve it by a two-stage process, as proposed by Gilmore and Gomory (1961).

Two Stage Cutting-Stock Algorithm

Input: Parameters, m, L, and the numbers b_i and ℓ_i for $i = 1, ..., m$.

Initialize: Set $\mathcal{C}$ to include only the m trivial cutting patterns (i.e., the ith one including only pieces of length ℓ_i, as many as possible).

Main Loop:

Stage One: Solve (5) with the current set of cutting patterns. Let $x^* = (x_c^* \mid c \in \mathcal{C})$ be an optimal solution, and $y^* = (y_1^*, ..., y_m^*)$ be an optimal dual solution.

Stage Two: Solve the integer knapsack problem

$$\begin{array}{lrll} \max & \sum_{i=1}^m y_i^* z_i & & \\ s.t. & \sum_{i=1}^m \ell_i z_i & \leq L & \\ & z_i & \in \mathbb{Z}_+ \text{ for all } i = 1, ..., m, & \end{array}$$

and let Z_{KP} be the optimum value.

Checking: If $Z_{KP} \leq 1$ then STOP (x^* is optimal in (5)), otherwise add $z = (z_1, ..., z_m)$ to $\mathcal{C}$ and RETURN to Stage One.

Output: c and x_c^* for all $c \in \mathcal{C}$ for which $x_c^* > 0$.

Proof. See class for proof of correctness ... □

Of course, we have

$$Z_{LP} \leq Z_{IP}$$

and the solution $(\lceil x_c^* \rceil \mid c \in \mathcal{C})$ is a feasible solution of the integer programming problem (4) for which we have

$$\sum_{c \in \mathcal{C}} \lceil x_c^* \rceil \leq Z_{LP} + m \leq Z_{IP} + m.$$

(WHY ... $+m$???)

This implies that whenever the number of long rods needed is much more than the number of different order lengths, then this is quite a good approximation!