MATH 409 LECTURES 19-21
THE KNAPSACK PROBLEM

REKHA THOMAS

We now leave the world of discrete optimization problems that can be solved in polynomial time and look at the easiest case of an integer program, called the knapsack problem.

The Knapsack Problem.
Given: $c_1, c_2, \ldots, c_n, w_1, w_2, \ldots, w_n$ and W all nonnegative integers.
Find: a subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in S} w_j \leq W$ and $\sum_{j \in S} c_j$ is maximum.

The physical interpretation is that we have a knapsack that can carry a total weight of W and can be filled with n different items where the j-th item has weight w_j and value c_j. We would like to find a collection of items to put into the knapsack so that the total weight of the knapsack is not exceeded and the total value of the knapsack is maximized.

The knapsack problem can be written as a 0/1 integer program as follows.

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} & \sum w_j x_j \leq W \\ & x_j = 0, 1 \ \forall \ j = 1, \ldots, n \end{array}$$

Relaxing the 0/1 constraint on the variables, we get the linear programming relaxation of the knapsack problem that is usually called the **fractional knapsack problem**.

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} & \sum w_j x_j \leq W \\ & 0 \leq x_j \leq 1 \ \forall \ j = 1, \ldots, n \end{array}$$

This linear program has a very easy solution that we show below.

Proposition 1. *(Dantzig 1957) Order the variables so that*

$$\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \cdots \geq \frac{c_n}{w_n}$$

Date: May 19, 2010.

and let

$$k := \min \{j \in \{1, \ldots, n\} : \sum_{i=1}^{j} w_i > W\}.$$

Then the optimal solution of the fractional knapsack problem is

$$x_1 = 1,\, x_2 = 1,\, \ldots,\, x_{k-1} = 1,\, x_k = \frac{W - \sum_{j=1}^{k-1} w_j}{w_k},\, x_j = 0 \;\forall\; j > k.$$

Note that $\frac{c_j}{w_j}$ is the value per unit length of the j-th item. So we first order the items in decreasing order of value per unit length and the optimal solution of the fractional knapsack problem is gotten by simply putting items in this order from the most valuable down until we cannot fit the next item, at which point we cut this last item to fit.

Exercise 2. Argue that Dantzig's proposed solution is indeed an optimal solution to the fractional knapsack problem. (First check that the proposed solution is indeed a feasible solution and then argue that no other solution has higher value for $\sum_{j=1}^{n} c_j x_j$.)

We now turn to the original knapsack problem. First note that we can assume that $w_j \leq W$ for all $j = 1, \ldots, n$ since otherwise, the j-th item will not fit in the knapsack and we would never include it in any solution making $x_j = 0$ always. As a start, we show that it is very easy to come up with a feasible solution of the knapsack problem whose objective function value is at least half the optimal value of the knapsack problem. This is an example of an **approximation algorithm** where one is typically interested in finding solutions to hard problems that come within a guaranteed factor of the optimal solution in value and where this feasible solution can be found quickly (in polynomial time).

Proposition 3. *Suppose $w_j \leq W$ for all $j = 1, \ldots, n$,*

$$\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \cdots \geq \frac{c_n}{w_n}$$

and

$$k := \min \{j \in \{1, \ldots, n\} : \sum_{i=1}^{j} w_i > W\}.$$

Then the better of the two solutions:

$$s_1 : (x_1 = 1,\, x_2 = 1,\, \ldots,\, x_{k-1} = 1,\, x_j = 0 \;\forall\; j \geq k)$$
$$s_2 : (x_k = 1,\, x_j = 0 \;\forall\; j \neq k)$$

is a feasible solution of the knapsack problem whose objective function value is at least half the optimal value of the knapsack problem.

Proof. From the optimal solution to the fractional knapsack problem, note that $C := \sum_{j=1}^{k} c_j$ is an upper bound on the optimal value of the fractional knapsack problem and hence also for the knapsack problem. Now note that C is the sum of the objective function value of s_1 and the objective function value of s_2. Therefore, at least one of the summands is half of C. This means that the better solution (with respect to $\sum_{j=1}^{n} c_j x_j$) has objective function value at least half of the optimal value of the knapsack problem. □

Exercise 4. Let $G = (V, E)$ be a graph and suppose we are interested in the problem of finding a cut in G of largest size. This means that we want a partition of V into two sets S and $V \backslash S$ such that the edges that leave S and enter $V \backslash S$ are as many as possible. This is a very difficult problem in discrete optimization called the **max cut problem** in G. However, it is not hard to come up with a cut in the graph that is guaranteed to have at least half as many edges as the max cut. This exercise walks you through the approximation procedure.

Suppose you have two colors red and blue to color the vertices of G. The coloring is done as follows. Start at any vertex and color it one of the two colors. Call this vertex v_1. To color the i-th vertex, we use the following rule: let b_i be the number of neighbors of v_i colored blue and r_i be the number of neighbors of v_i colored red. If $b_i \geq r_i$ then color v_i red and otherwise, color v_i blue.

Argue that at the end, at least half the edges in the graph have the property that one of their end points is red and the other is blue. This produces a cut in G (induced by the partition of the vertices into red vertices and blue vertices) of size at least half the total number of edges in the graph.

We now write down an algorithm to compute the optimal solution of the knapsack problem.

Dynamic Programming Knapsack Algorithm

Let C be an upper bound on the optimal value of the knapsack problem. For instance take $C := \sum_{j=1}^{n} c_j$.

(1) Set $x(0,0) := 0$ and $x(0,k) := \infty$ for all $k = 1, \ldots C$.

(2) For $j = 1, \ldots, n$ do
    For $k = 0, \ldots, C$ do
        set $s(j,k) := 0$ and $x(j,k) := x(j-1,k)$
    For $k = c_j, \ldots, C$ do

If $x(j-1, k-c_j) + w_j \leq \min\{W, x(j,k)\}$ then
set $x(j,k) := x(j-1, k-c_j) + w_j$ and $s(j,k) := 1$

(3) Let $k = \max\{i \in \{0, \ldots, C\} : x(n,i) \text{ is finite }\}$. Set $S := \emptyset$.
For $j = n, \ldots, 1$ do
If $s(j,k) = 1$ then set $S := S \cup \{j\}$ and $k := k - c_j$.

The optimal solution of the knapsack problem is given by choosing the items in S.

Exercise 5. Run the above algorithm on the following problem.

$$\begin{array}{ll} \max & 3x_1 + 2x_2 + x_3 + x_4 + x_5 \\ \text{s.t.} & x_1 + 2x_2 + 3x_3 + 4x_4 + 5x_5 \leq 13 \\ & x_1, \ldots, x_5 = 0, 1 \end{array}$$

Theorem 6. *The dynamic programming algorithm finds an optimal solution to the knapsack problem in $O(nC)$ time.*

Proof. The algorithm takes $O(nC)$ steps just by looking at all the for loops.

To prove the algorithm, we interpret the variable $x(j,k)$ as the minimum total weight of a subset $S \subseteq \{1, \ldots, j\}$ with $\sum_{i \in S} c_i = k$. Check that this is true in the example you did. The algorithm computes these values by the following recursion:

$$x(j,k) = \begin{cases} x(j-1, k-c_j) + w_j & \text{if } c_j \leq k \text{ and } x(j-1,k-c_j) + w_j \\ & \leq \min\{W, x(j-1,k)\} \\ x(j-1,k) & \text{otherwise} \end{cases}$$

for $j = 1, \ldots, n$ and $k = 0, \ldots, C$. The variable $s(j,k)$ indicates which of the two cases happen.

The algorithm enumerates all subsets $S \subseteq \{1, \ldots, n\}$ that are feasible and not dominated by others. A set S is dominated by a set S' if $\sum_{j \in S} c_j = \sum_{j \in S'} c_j$ and $\sum_{j \in S} w_j \geq \sum_{j \in S'} w_j$. In step (3) the best feasible S is chosen.

□

Note that the size of the input to the knapsack problem is $O(n \log C + n \log W)$ and that $O(nC)$ is not polynomial in the size of the input.

Definition 7. Let $\mathcal{P}$ be a decision problem or an optimization problem such that each instance x consists of a list of integers. Denote by $largest(x)$ the largest of these integers. An algorithm for $\mathcal{P}$ is called **pseudopolynomial** if its running time is bounded by a polynomial in $size(x)$ and $largest(x)$.

The dynamic programming algorithm mentioned above is a pseudopolynomial time algorithm since nC is a quadratic polynomial in n and C and n is part of the input size and C is bounded above by $largest(x)$. A second example of a pseudopolynomial time algorithm is the algorithm to test for the primality of an integer p by dividing p with all integers from $2, \ldots, \lfloor \sqrt{p} \rfloor$.