Knapsack Problem

Zdeněk Hanzálek
zdenek.hanzalek@cvut.cz

CTU in Prague

March 31, 2020

Table of contents

Problem Formulation

Knapsack problem

- **Instance:** Nonnegative integers $n, c_1, \ldots, c_n, w_1, \ldots, w_n, W$, where n represents the number of items, $c_1, \ldots, c_n$ represents the cost of each item, $w_1, \ldots, w_n$ represents the weight of each item and W is the maximum weight to be carried in the knapsack.
- **Goal:** 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 the maximum.

- It is one of the "easiest" NP-hard problems.
- Sometimes it is called **0/1 Knapsack problem**.

Fractional Knapsack Problem

While relaxing on the indivisibility of each item, we formulate a new problem:

Fractional Knapsack problem

- **Instance:** Nonnegative integers $n, c_1, \ldots, c_n, w_1, \ldots, w_n, W$, where n represents the number of items, $c_1, \ldots, c_n$ represents the cost of each item, $w_1, \ldots, w_n$ represents the weight of each item and W is the maximum weight to be carried in the knapsack.
- **Goal:** Find the rational numbers $x_1, \ldots, x_j, \ldots, x_n$ such that $0 \leq x_j \leq 1$ and $\sum_{j=1}^{n} x_j \cdot w_j \leq W$ and $\sum_{j=1}^{n} x_j \cdot c_j$ is the maximum.

- Since the items can be divided (continuous variable x_j), we can solve this problem in polynomial time.

Solution of Fractional Knapsack Problem - Dantzing [1957]

- if $\sum_{j=1}^{n} w_j > W$ (otherwise it has a trivial solution)
- order and re-index the items by their relative cost: $\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \ldots \geq \frac{c_n}{w_n}$
- in the ordered sequence find the first item which does not fit in the knapsack ($h := min\left\{j \in \{1,\ldots,n\} : \sum_{i=1}^{j} w_i > W\right\}$)
- in order to find the optimal solution, we cut off a part of h-th item, so that this part fits in the knapsack:
  - $x_j := 1$ for $j = 1,\ldots,h-1$
  - $x_h := \frac{W - \sum_{i=1}^{h-1} w_i}{w_h}$
  - $x_j := 0$ for $j = h+1,\ldots,n$
- sorting of the items takes $O(nlogn)$, computing h can be done in $O(n)$ by simple linear scanning, so this algorithm solves the **Fractional Knapsack problem** in $O(nlogn)$
- there is an even faster algorithm ([1] page 440) which can solve this problem in $O(n)$ by reduction to the **Weighted Median problem**

2-Approximation Algorithm for Knapsack

r-approximation algorithm for maximization

Algorithm A for objective function J maximization is called r-approximation if there exists a number $r \geq 1$ such that $J^A(I) \geq \frac{1}{r} J^*(I)$ for all instances I of this problem.

Theorem

Let $n, c_1, \ldots, c_n, w_1, \ldots, w_n, W, h$ are nonnegative integers that satisfy:

- $w_j \leq W$ pro $j = 1, \ldots, n$
- $\sum_{i=1}^{n} w_i > W$
- $\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \ldots \geq \frac{c_n}{w_n}$
- $h = min\left\{j \in \{1, \ldots, n\} : \sum_{i=1}^{j} w_i > W\right\}$

Then choosing the better of the two solutions $\{1, \ldots, h-1\}$ or $\{h\}$ is **2-approximation alg. for the Knapsack** with the time complexity $O(n)$.

Example: $c = (6, 20, 2), w = (2, 10, 8), W = 10$.

2-Approximation Algorithm for Knapsack

Proof:

- Given any instance of the **Knapsack problem**, we can omit all items whose weight is bigger than the knapsack maximum load.
- If $\sum_{i=1}^{n} w_i \leq W$ then the whole set of items is an optimal solution.
- Since $\sum_{i=1}^{h} c_i$ is an upper bound on the optimum value, the better of two solutions $\{1, \ldots, h-1\}$ and $\{h\}$ achieves at least half of the optimum value, e.i., $J^A(I) = max\left\{\sum_{i=1}^{h-1} c_i, c_h\right\} \geq \frac{\sum_{i=1}^{h} c_i}{2} > \frac{1}{2} J^*(I)$

Notes about approximation algorithms:

- An approximation algorithm guarantees, that even in the worst case, the value of the objective function will be proportional to the optimum value. The frequency of the worst case is not considered by the approximation algorithm.
- ϵ, the relative deviation from the optimum, is sometimes used instead of the asymptotic performance ratio r, so that $r = 1 + \epsilon$.
- In this case, it is meaningless to state the absolute error. Why?

Dynamic Programming (Integer Costs) for Knapsack

- Pseudopolynomial algorithm with time complexity $O(nC)$.
- Variable x_k^j represents the **minimum weight with cost k which can be achieved as a selection of items from set** $\{1, \ldots, j\}$
- (*) Item j is added to the selection of items from $1, \ldots, j$ if for the given price k this set reaches the **lower or equal weight as set** $1, \ldots, j-1$.
  The algorithm computes these values using the recursion formula:

$$x_k^j = \begin{cases} x_{k-c_j}^{j-1} + w_j & \text{if item } j \text{ was added;} \\ \\ x_k^{j-1} & \text{if item } j \text{ wasn't added.} \end{cases}$$

- In variable s_k^j we memorize which of the two possible cases has happened. It is later used to reconstruct the selection.

Blackboard example (integer costs):
$n = 4, w = (21, 35, 52, 17), c = (10, 20, 30, 10), W = 100.$

Dynamic Programming (Integer Costs) for Knapsack

Input: Costs $c_1, \ldots, c_n \in \mathbb{Z}_0^+$, weights $w_1, \ldots, w_n, W \in \mathbb{R}_0^+$.
Output: $S \subseteq \{1, \ldots, n\}$; $\sum_{j \in S} w_j \leq W$ and $\sum_{j \in S} c_j$ is maximum.

Let C be the arbitrary upper bound of the solution, e.g. $C = \sum_{j=1}^{n} c_j$;
$x_0^0 := 0$; $x_k^0 := \infty$ for $k = 1, \ldots, C$;
for $j := 1$ **to** n **do**
  for $k := 0$ **to** C **do** $x_k^j := x_k^{j-1}$; $s_k^j := 0$;
  for $k := c_j$ **to** C **do**
    if $x_{k-c_j}^{j-1} + w_j \leq min\{W, x_k^{j-1}\}$ **then**
      $x_k^j := x_{k-c_j}^{j-1} + w_j$; $s_k^j := 1$;
    end
  end
end
$i := max\{k \in \{0, \ldots, C\} : x_k^n < \infty\}$; $S := \oslash$;
for $j := n$ **downto** 1 **do**
  if $s_i^j = 1$ **then** $S := S \cup \{j\}$; $i := i - c_j$;
end

Dynamic programming for Knapsack - Overview

Dynamic programming overview

- Pseudopolynomial Algorithm
- State space (may be represented by a graph) is constructed due to **integer weights or costs** of items.
- Due to the **optimal substructure** property of the problem, the optimal solution may be found by the **recurrent formula** using **Bellman's Principle of Optimality**.
- The problem has **overlapping subproblems**. State space is not the tree, it contains **diamonds**, places in the state space where we keep only the better of two possible solutions - refer to item (*) - prevents exponential growth of the state space size.

If the weights are integers, we can solve the problem by dynamic programming while selecting the solution having the **higher** cost for given weight (*) while initializing $x_k^0 := -\infty$ for $k = 1, \ldots, C$.
Blackboard example (integer weights):
$n = 4, w = (2, 3, 4, 5), c = (3.1, 4.2, 5.1, 4.3), W = 8$.

Complexity Reduction by Rounding Data

The time complexity of the dynamic programming algorithm for knapsack depends on C.

Idea

Divide all costs $c_1, \ldots, c_n$ by 2 and round them down.

The algorithm becomes faster, but we can obtain a suboptimal solution. This allows us to find a tradeoff between the speed and desired optimality. By

$$\bar{c}_j := \left\lfloor \frac{c_j}{t} \right\rfloor \quad \textit{for } j = 1, \ldots, n$$

the time complexity of the Dynamic programming algorithm for Knapsack is reduced t-times.

Approximation Scheme for Knapsack

Input: Nonnegative integer numbers $n, c_1, \ldots, c_n, w_1, \ldots, w_n, W$. Number $\epsilon > 0$.

Output: Subset $S \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in S} w_j \leq W$ and $\sum_{j \in S} c_j \geq \frac{1}{1+\epsilon} \sum_{j \in U} c_j$ for all $U \subseteq \{1, \ldots, n\}$ satisfying $\sum_{j \in U} w_j \leq W$.

1. run **2-approximation algorithm for Knapsack**;
   label the solution as S_1 with cost $c(S_1) = \sum_{j \in S_1} c_j$;
2. $t := max\{1, \frac{\epsilon c(S_1)}{n}\}$;
   $c'_j := \left\lfloor \frac{c_j}{t} \right\rfloor$ *for* $j = 1, \ldots, n$;
3. Run the **Dynamic programming algorithm for Knapsack** with instance $(n, c'_1, \ldots, c'_n, w_1, \ldots, w_n, W)$ using the upper bound $C := \frac{2c(S_1)}{t}$;
   label the solution as S_2 with cost $c(S_2) = \sum_{j \in S_2} c_j$;
4. **if** $c(S_1) > c(S_2)$ **then** $S := S_1$;
   else $S := S_2$;

Approximation Scheme for Knapsack

- **Knapsack** is one of a few problems whose approximation algorithm can have an arbitrary small ϵ, i.e. relative deviation form the optimum.
  - The choice of $t := \frac{\epsilon c(S_1)}{n}$ leads to $(1+\epsilon)$-approximation algorithm, but we do not show the proof of this statement.
  - If ϵ is too small, i.e. $\epsilon \leq \frac{n}{c(S_1)}$, then $t = 1$, i.e. we find optimal solution while using the **Dynamic programming algorithm for Knapsack** with the upper bound from the 2-approximation algorithm for Knapsack.
- Time complexity is $O(nC) = O(n\frac{c(S_1)}{t}) = O(n\frac{c(S_1)n}{\epsilon c(S_1)}) = O(n^2 \cdot \frac{1}{\epsilon})$.

Knapsack - Summary

- One of the "easiest" NP-hard problems
- Basic problem for many other optimization problems (bin packing, container loading, 1-D, 2-D, 3-D cutting problem)

References


B. H. Korte and Jens Vygen.
Combinatorial Optimization: Theory and Algorithms.
Springer, fourth edition, 2008.