0-1 Knapsack solution

Sadrtdinov Ildus

March 2020

1 Introduction

This paper describes an approximate solution for 0-1 Knapsack problem written in Rust. This solution consists of two parts, **Ibarra-Kim** algorithm, which is memory inefficient, and **Heuristic** algorithm, which is long in time. The combination of these two algorithms provides a powerful solution, which deals successfully with many cases and even strikes an optimal dynamic solution for some tests. The solution simply chooses the best answer from two mentioned algorithms.

This notation will be used further:

- W - the capacity of the knapsack
- n - number of items
- $\{(c_i, w_i)\}_{i=1}^{n}$ - items to full the knapsack with costs c_i and weights w_i
- C_{greedy} - greedy solution cost

2 Ibarra-Kim algorithm

This algorithm outputs a $\frac{1}{\varepsilon}$-approximate solution for $0 < \varepsilon < \frac{1}{2}$. The approach of this method is to rescale item costs to range $\left[0, \frac{1}{\varepsilon^2}\right]$ with a simple formula:

$$c_i^* = \frac{c_i}{2\varepsilon^2 C_{greedy}}$$

and to launch dynamic programming (DP) for rescaled costs. This results in **time complexity** of $\mathcal{O}\left(\frac{n}{\varepsilon^2}\right)$.

It turned out that DP with reduced memory doesn't work for costs (in the way that it works for weights). For this reason, DP maintains the full dynamic table, which results in $\mathcal{O}\left(\frac{n}{\varepsilon^2}\right)$ **memory cost**.

In final implemention ε is set as follows:

$$\varepsilon = \begin{cases} 0.0195, & n \leq 50000 \\ 0.04, & n > 50000 \end{cases}$$

3 Heuristic algorithm

The heuristic algorithm is based on the greedy solution. The idea of the algorithm is rather simple: we definitely should take items with the greatest unit cost (the same as the greedy solution does) and we also should consider a list of candidates, which would fill the remaining space of the knapsack.

Denote item unit cost by $u_i = \frac{c_i}{w_i}$. Let $\sigma \in S_n$ be a permutation, which sorts items by unit cost, i.e. $u_{\sigma(1)} \geq u_{\sigma(2)} \geq \ldots u_{\sigma(n)}$. We will take two parameters: $\alpha \in \mathbb{R}, 0 \leq \alpha \leq 1$ and $m \in \mathbb{N}, m \leq n$. The algorithm consists of several steps:

1. Fill αW of knapsack's capacity with items with the greatest unit cost, i.e. find j such that $\sum_{i=1}^{j} w_{\sigma(j)} \geq \alpha W$. Items $\sigma(1), \ldots, \sigma(j-1)$ are taken to the knapsack.

2. Let $W_{new} = W - \sum_{i=1}^{j-1} w_{\sigma(j)}$. Consider items $\sigma(j), \ldots, ..., \sigma(j+m-1)$. Solve 0-1 knapsack problem for these items and capacity W_{new}.

The last step is done by DP with reduced memory for weights.
Time complexity: $\mathcal{O}\left(mW_{new}\right) = \mathcal{O}\left((1-\alpha)mW\right)$
Memory cost: $\mathcal{O}\left(W_{new}\right) = \mathcal{O}\left((1-\alpha)W\right)$

Values for parameters used in the final implementation: $\alpha = 0.5, m = 1200$.

4 Results

The solution was tested in Yandex Contest.
Time limit - 10s, **memory limit** - 512 Mb.

№	n	W	**Greedy score**	**DP score**	**Solution score**	**Time**	**Memory**
1	50000	1892492	29433040	29448192	29448082	7.858s	30.50Mb
2	100000	577535	13472213	13475909	13475909	2.414s	11.76Mb
3	50000	1468889	11170413	11179617	11179616	6.079s	24.04Mb
4	100000	543896	4876185	4877302	4877300	2.286s	10.55Mb
5	50000	1237528	799761	1117491	1117344	3.258s	466.03Mb
6	100000	326301	199953	296293	296279	1.215s	207.24Mb
7	50000	2243398	33791873	33792261	33792261	9.166s	31.16Mb
8	100000	701824	12881119	12883188	12883188	2.9s	10.88Mb
9	50000	477920	528973	538688	538688	2.919s	508.70Mb
10	200000	198956	173206	214214	214213	1.673s	492.90Mb