An Algorithmic Approach to Knapsack Problems

Lily Zhang

Macalester College
Fall 2022 – Professor Kristin Heysse

Abstract

The knapsack problem is a classical combinatorial optimization problem: select items to maximize value without exceeding a knapsack's capacity. As the capacity or the number of item types grows, the problem size and computational complexity increase rapidly. Thus, it has been widely studied and approached using various algorithms. In this paper, we present algorithms for the $0-1$ knapsack problem, the fractional knapsack problem, and the multidimensional knapsack problem, and we illustrate how these methods can be applied in real-life practice.

1 Overview

1.1 What Are Knapsack Problems?

Suppose we are planning a hiking trip and are therefore interested in filling a knapsack with items considered necessary for the trip. There are n different item types that are deemed desirable, such as a bottle of water, an apple, an orange, or a sandwich. Each item type has two attributes: a weight (or volume) and a value that quantifies the level of importance associated with each unit of that type of item. Since the knapsack has a limited weight (or volume) capacity, the problem of interest is figuring out how to load the knapsack with a combination of units of the specified types of items that yields the greatest total value [5].

The problem discussed so far is the prototypical knapsack problem in which a set of items is given, each with an associated value and weight. The goal is to select a subset such that the total weight does not exceed a given bound and the total value is maximized.

1.2 Mathematical Formulation

The knapsack problem can be mathematically formulated [3] by numbering the items from 1 to n and introducing a vector of binary variables x_j $(j = 1, \ldots, n)$ having the following meaning:

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

Then, if p_j is the profit (value) of item j, w_j its weight and c the capacity of the knapsack, our problem will be to select, from among all binary vectors x satisfying the constraint

$$\sum_{j=1}^{n} w_j x_j \leq c,$$

the one which maximizes the objective function

$$\sum_{j=1}^{n} p_j x_j.$$

1.3 Terminology

The objects considered in the previous section will generally be called *items* and their number be indicated by n. The value and size associated with the jth item will be called *profit* and *weight*, respectively, and denoted by p_j and w_j ($j \in \{1, \ldots, n\}$).

It is always assumed, as is usual in the literature, that profits, weights and capacities are positive values. The results obtained, however, can easily be extended to the case of real values and, in the majority of cases, to that of **non-negative** values.

The prototype problem of the previous section,

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & x_j = 0 \text{ or } 1, \\ & j \in \{1, \ldots, n\} \end{aligned}$$

is known as the 0–1 *Knapsack Problem* and will be analyzed in Section 2.

In Section 3 we consider the generalization arising when we no longer demand a solution to be integral but allow for fractional values; this yields the *Fractional Knapsack Problem*. Another important generalization of the 0–1 knapsack problem, in Section 4, is the *Multidimensional Knapsack Problem*, arising when there is more than one capacity constraint available.

2 0–1 Knapsack Problem

2.1 Problem Statement

The 0–1 *Knapsack Problem* (KP) is [4]: given a set of n *items* and a *knapsack*, with

$$\begin{aligned} p_j &= \textit{profit} \text{ of item } j, \\ w_j &= \textit{weight} \text{ of item } j, \\ c &= \textit{capacity} \text{ of the knapsack}, \end{aligned}$$

select a subset of the items so as to

$$\begin{aligned} \text{maximize} \quad & z = \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & x_j = 0 \text{ or } 1, \quad j \in \{1, \ldots, n\} \end{aligned}$$

where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

As usual, we discard items with $w_j > c$, so we may assume $w_j \leq c$ for all j.

2.2 Algorithms

There are several algorithms that can be used to solve the 0–1 knapsack problem, including dynamic programming, approximation, and greedy algorithms. Dynamic programming algorithms involve breaking the problem down into smaller subproblems and solving these subproblems recursively. Approximation algorithms provide a solution that is close to the optimal solution, but may not be the optimal solution itself. Greedy algorithms make a locally optimal choice at each step in the hope of finding a globally optimal solution.

2.2.1 Dynamic Programming Algorithm

Algorithm 1 Dynamic Programming Knapsack Algorithm [2]

Require: Nonnegative integers n, $c_1, \ldots, c_n$, $w_1, \ldots, w_n$, W (*here c_j are profits, w_j weights, W capacity*)

```
C ← Σ_{j=1}^{n} c_j
x(0,0) ← 0
x(0,k) ← ∞ for k = 1, ..., C
for j ← 1 to n do
  for k ← 0 to C do
    s(j,k) ← 0
    x(j,k) ← x(j−1,k)
  end for
  for k ← c_j to C do
    if x(j−1,k−c_j) + w_j ≤ min(W, x(j,k)) then
      x(j,k) ← x(j−1,k−c_j) + w_j
      s(j,k) ← 1
    end if
  end for
end for
```

2.2.2 Approximation Algorithm

Algorithm 2 Knapsack Approximation Scheme (FPTAS) [2]

Require: Nonnegative integers n, $c_1, \ldots, c_n$, $w_1, \ldots, w_n$, W with $w_j \le W$; a number $\epsilon > 0$. {Output: $S \subseteq \{1, \ldots, n\}$ with $\sum_{j \in S} w_j \le W$ and $c(S) \ge (1-\epsilon) \cdot \mathrm{OPT}$.}

Let $\frac{c_1}{w_1} \ge \frac{c_2}{w_2} \ge \cdots \ge \frac{c_n}{w_n}$.

$S_1 \leftarrow \min\{j \in \{1, \ldots, n\} : \sum_{i=1}^{j} w_i > W\}$ (*greedy prefix index*)

if $c(S_1) = 0$ **then**
  $S \leftarrow S_1$; **stop**
end if

$t \leftarrow \max\left\{1, \ \frac{\epsilon\, c(S_1)}{n}\right\}$

for $j = 1$ **to** n **do**
  $c_j \leftarrow \lfloor c_j / t \rfloor$
end for

Apply the DP algorithm to $(n, c_1, \ldots, c_n, w_1, \ldots, w_n, W)$ with profit cap $C \leftarrow \frac{2\, c(S_1)}{t}$, and let S_2 be the returned solution.

if $c(S_1) > c(S_2)$ **then**
  $S \leftarrow S_1$
else
  $S \leftarrow S_2$
end if

Theorem 1. (Greedy Algorithm Redux) *Sort items by nonincreasing p_j/w_j and take the maximal prefix that fits (greedy-by-ratio). Return the better of: (i) this greedy prefix and (ii) the single best item that fits. This is a* $1/2$*-approximation for the* 0–1 *knapsack problem.*

Proof. Order items so that $\frac{p_1}{w_1} \geq \cdots \geq \frac{p_n}{w_n}$. Let k be the first index that does not fit after taking items $1, \ldots, k-1$ greedily, and let V_G be the total profit of this greedy prefix. The optimal *fractional* value satisfies

$$\text{OPT} \;\leq\; V_{\text{frac}} \;=\; V_G \;+\; \Big(W - \sum_{j<k} w_j\Big)\frac{p_k}{w_k} \;\leq\; V_G + p_k.$$

Since we discard items with $w_j > W$, we have $w_k \leq W$, hence $p_k \leq V_*$ where V_* is the profit of the single best item that fits. Therefore $\text{OPT} \leq V_G + V_*$. It follows that $\max\{V_G, V_*\} \geq \frac{1}{2}\,\text{OPT}$, which is exactly the claimed $1/2$-approximation guarantee. □

3 Fractional Knapsack Problem

3.1 Problem Statement

Relaxing the 0–1 constraint on the variables, the *Fractional Knapsack Problem* (FKP) is [4]: given a set of n *items* and a *knapsack*, with

$$\begin{aligned} p_j &= \textit{profit} \text{ of item } j, \\ w_j &= \textit{weight} \text{ of item } j, \\ c &= \textit{capacity} \text{ of the knapsack}, \end{aligned}$$

select a subset of the items so as to

$$\begin{aligned} \text{maximize} \quad & z = \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_j x_j \leq c, \\ & 0 \leq x_j \leq 1, \\ & j \in \{1, \ldots, n\}. \end{aligned}$$

3.2 Greedy Algorithm and Optimality

In contrast to the 0–1 knapsack problem, the fractional knapsack problem is solved optimally by a simple greedy algorithm: consider the items in decreasing value-to-weight ratio and add whole items to the knapsack one at a time until we reach an item whose addition would exceed the capacity c. Then, add the largest fraction of that item that fits and stop.

Algorithm 3 Fractional Knapsack Algorithm [2]

Sort the items so that $\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}$
$s \leftarrow 0$ (*current total weight*)
$k \leftarrow 1$
while $k \leq n$ **and** $s + w_k \leq c$ **do**
  $x_k \leftarrow 1$
  $s \leftarrow s + w_k$
  $k \leftarrow k + 1$
end while
if $k \leq n$ **then**
  $x_k \leftarrow \dfrac{c - s}{w_k}$
  for $t = k + 1$ **to** n **do**
    $x_t \leftarrow 0$
  end for
end if

3.3 Proof of Optimality

Without loss of generality, assume the items are indexed in the sorted order:

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n}.$$

Let $V(S) := \sum_{t=1}^{n} p_t x_t$ denote the value of a (possibly fractional) solution $S = (x_1, \ldots, x_n)$. If $\sum_{t=1}^{n} w_t \leq c$, the optimal solution is $(1, 1, \ldots, 1)$, which the algorithm returns. In the remainder, assume $\sum_{t=1}^{n} w_t > c$.

Lemma 2. *Let $S = (x_1, \ldots, x_n)$ be a feasible solution and let $i < j$ with $x_i < 1$ and $x_j > 0$. Then there exists a feasible $S' = (x'_1, \ldots, x'_n)$ with $V(S') \geq V(S)$, identical to S except possibly at i and j, and either $x'_i = 1$ or $x'_j = 0$.*

Proof. Transfer weight from item j (lower ratio) to item i (higher ratio). There are two cases.

Case 1. If $w_j x_j \leq w_i(1 - x_i)$ (we can transfer all weight of j), set

$$x'_t = x_t \ (t \neq i, j), \qquad x'_i = x_i + \frac{w_j}{w_i} x_j, \qquad x'_j = 0.$$

Feasibility: $0 \leq x'_i \leq x_i + (1 - x_i) = 1$ since $\frac{w_j}{w_i} x_j \leq 1 - x_i$ by the hypothesis; all other bounds are clear, and the total weight is unchanged. Value:

$$V(S') - V(S) = p_i\Big(\tfrac{w_j}{w_i} x_j\Big) - p_j x_j = \Big(\tfrac{p_i}{w_i} - \tfrac{p_j}{w_j}\Big) w_j x_j \geq 0.$$

Case 2. If $w_j x_j > w_i(1 - x_i)$ (we fill item i before exhausting j), set

$$x'_t = x_t \ (t \neq i, j), \qquad x'_i = 1, \qquad x'_j = x_j - \left(\frac{w_i}{w_j}\right)(1 - x_i).$$

Feasibility: $x'_j \in [0, 1]$ by the case condition, total weight is unchanged, and $x'_i = 1$. Value:

$$V(S') - V(S) = p_i(1 - x_i) - p_j\left(\frac{w_i}{w_j}\right)(1 - x_i) = \left(\frac{p_i}{w_i} - \frac{p_j}{w_j}\right) w_i(1 - x_i) \geq 0.$$

□

Lemma 3. *If $S = (x_1, \ldots, x_n)$ is an optimal solution, then $\sum_{t=1}^{n} w_t x_t = c$.*

Proof. If $\sum_{t=1}^{n} w_t x_t < c$, choose any k with $x_k < 1$ (exists because $\sum_{t=1}^{n} w_t > c$) and increase x_k to $x_k + \min\left(\frac{c - \sum w_t x_t}{w_k}, 1 - x_k\right)$, improving the value and preserving feasibility, contradicting optimality. □

Lemma 4. *Let $G = (x_1^g, \ldots, x_n^g)$ be the solution returned by Algorithm 3. Then there exists k such that $x_t^g = 1$ for all $t < k$, $x_k^g \in [0, 1]$, and $x_t^g = 0$ for all $t > k$, and $\sum_{t=1}^{n} w_t x_t^g = c$.*

Proof. Immediate from the loop condition and the final fractional assignment in Algorithm 3. □

Conclusion. Starting from any optimal solution, repeatedly apply Lemma 2 to eliminate anomalies (pairs $i < j$ with $x_i < 1$ and $x_j > 0$) without decreasing value; by Lemma 3 the optimal solution can be taken to be tight ($\sum w_t x_t = c$), and by the sorted ratios this anomaly-free, tight solution has the structure of Lemma 4. Therefore Algorithm 3 returns an optimal solution to the *fractional* knapsack problem [1].

4 Multidimensional Knapsack Problem

4.1 Problem Statement

The *Multidimensional Knapsack Problem* (MDKP) is [4]: given a set of n *items* and m *resource constraints (dimensions)*, with

$$\begin{aligned} p_j &= \textit{profit} \text{ of item } j, \\ w_{ij} &= \textit{weight/consumption} \text{ of item } j \text{ in resource } i, \quad i = 1, \ldots, m, \\ c_i &= \textit{capacity} \text{ of resource } i, \end{aligned}$$

the goal is to find a subset of items that maximizes the total profit while satisfying all resource constraints. Formally,

$$\begin{aligned} \text{maximize} \quad & z = \sum_{j=1}^{n} p_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} w_{ij} x_j \leq c_i, \quad i \in \{1, \ldots, m\} \\ & x_j = 0 \text{ or } 1, \quad j \in \{1, \ldots, n\} \end{aligned}$$

where

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

When $m = 1$, MDKP reduces to the 0–1 knapsack problem considered in Section 2.

5 Work Example

To illustrate how knapsack problems can be applied in real-life situations, we use two knapsack algorithms in two settings. This shows how these algorithms can support more rational decisions in practical scenarios.

Setting 1

During Black Friday, almost every product has a discount. As a clever customer, we want to maximize the discount within a given budget. We collect data from Amazon and design a problem about how to select the products we want from a total of 30 products to maximize the discount while not exceeding the $500 budget. In this setting, we introduce two algorithms: the 0–1 knapsack algorithm and an approximation algorithm. Table 1 shows the first 5 items of the 30 products; the full product list is shown in the Appendix as Table 6.

Item	**Discounted_Price**	**Original_Price**	**Discount**
Formal shoe	$23	$36.99	$14
Tablet	$30	$49.99	$20
Headphone	$60	$149.99	$90
Printer	$20	$59.99	$40
Speaker	$45	$59.99	$15

Table 1: First 5 Products Information

5.1 0–1 Knapsack Algorithm

We utilize a dynamic programming method to implement the 0–1 knapsack algorithm. To find the optimal result for picking the products, we

- Create a matrix with rows representing items and columns representing the remaining budget (capacity).
- Loop through the matrix and calculate the discount that can be obtained by each combination of items at each capacity.
- Examine the completed matrix to determine which products to buy in order to maximize the total discount.

In Python, we can create this matrix by:

```
dp = [[0 for _ in range(col+1)] for _ in range(row+1)]
```

We pad the rows and columns by 1 so that the indices match the item and weight numbers. Now that we have created our matrix, we fill it by looping over the rows and columns.

For each element, we compute its value. If the item at the index matching the current row fits within the weight capacity represented by the current column, we take the maximum of either:

- The total value of the items already in the knapsack, or
- The total value of all the items in the knapsack except the item at the previous row index, plus the new item's value.

In Python, we can implement the method by:

```
for item in range(1, row+1):
    for budget in range(col+1):
        if price[item-1] > budget:
            dp[item][budget] = dp[item-1][budget]
            continue
        previous_best_discount = dp[item-1][budget]
        new_best_discount = discount[item-1] + dp[item-1][budget - price[
                                                item-1]]
        dp[item][budget] = max(previous_best_discount, new_best_discount)
```

In other words, as our algorithm considers each item, we decide whether adding this item would produce a higher total value than the last choice, given the knapsack's current total weight. If the current item is a better choice, we include it; if not, we leave it out.

Item	Discounted_Price	Discount
Formal shoe	$23	$13
Headphone	$59	$90
Printer	$19	$40
Sneaker	$55	$34
Wireless speaker	$99	$100
Bluetooth headphone	$78	$51
Skateboard	$36	$22
PS controller	$38	$25
Shirt	$27	$12
Sunglass	$34	$55
Swimsuit	$31	$22

Table 2: Result from Dynamic Programming Method

Result

The result derived from the dynamic programming method is shown in Table 2. The total price is $499, and we save $465 through this purchasing match.

5.2 Approximation Algorithm

We can also employ an approximation algorithm here to approximate the optimal solution for this problem setting. We

- Sort the items by the discount obtained per dollar spent.
- Loop through the sorted items and add an item if it does not exceed the budget.
- If the next item would exceed the budget, compare the discount from buying just that item versus the discount from all previously picked items, and take the better of the two (modified greedy).

In Python, we can implement the method by:

```
def approx_alg(w, v, W):
    ordered = []
    num = 1
    for vi, wi in zip(v, w):
        ordered.append([vi/wi, vi, wi, num])
        num += 1
    ordered.sort(reverse=True)

    # greedy-by-ratio prefix
```

```
    S1 = []
    weight = 0
    val = 0
    for i in range(len(ordered)):
        vi, wi, itemNum = ordered[i][1], ordered[i][2], ordered[i][3]
        if weight + wi <= W:
            S1.append([itemNum, vi, wi])
            weight += wi
            val += vi

    # best single item that fits
    best = None
    best_val = 0
    for ratio, vi, wi, itemNum in ordered:
        if wi <= W and vi > best_val:
            best_val = vi
            best = [itemNum, vi, wi]

    # return the better of greedy prefix vs best single item
    if best is None or val >= best_val:
        return S1, val
    else:
        return [best], best_val
```

Result

The result derived from the approximation method is shown in Table 3. The total price is $485, and we save $463 through this purchasing match. Comparing Table 2 and Table 3, the approximation is close to the optimal result ($465 vs. $463).

Item	Discounted_Price	Discount
Printer	$19	$40
Sunglass	$34	$55
Headphone	$59	$90
Wireless speaker	$99	$100
Swimsuit	$31	$22
Tablet	$29	$20
PS controller	$38	$25
Bluetooth headphone	$78	$51
Sneaker	$55	$34
Skateboard	$36	$22

Table 3: Result from Approximation Method

Setting 2

In the magic potion shop, we can purchase different kinds of potions, including the Elixir of Power, Wisdom, Healing, Magic, Invisibility, Strength, and Agility. Our aim is to bring them back to our hometown to sell in order to make a profit. However, considering the limited capacity of our backpack, which is 200, and the varying profit for each potion, we need to find the best matching of potions. The weights and profits for each potion are shown in Table 4.

Potion Type	**Power**	**Wisdom**	**Healing**	**Magic**	**Invisibility**	**Strength**	**Agility**
Weight	50	60	30	80	50	20	70
Profit	$340	$100	$180	$120	$220	$300	$60

Table 4: Potion Table for Setting 2

Here, we introduce the fractional knapsack algorithm to solve this problem. We:

- Sort the potions according to the profit per unit of capacity.
- Take whole items greedily, then (if needed) a final partial item.
- Obtain the optimal solution.

In Python, we can implement the method by:

```
items_list.sort(key=lambda it: it.value / it.weight, reverse=True)

for item in items_list:
    if capacity >= item.weight:
        capacity -= item.weight
        knapsack_value += item.value
        knapsack.append(item.item)
        knapsack_weights.append(item.weight)
    else:
        fraction = capacity / item.weight
        knapsack_value += item.value * fraction
        knapsack.append(item.item)
        knapsack_weights.append(round(fraction, 2))
        capacity = 0
        break
```

Result

The result derived from the greedy algorithm is shown in Table 5. The total profit is $1,123.33 using total weight 200.

Potion Type	Power	Wisdom	Healing	Magic	Invisibility	Strength	Agility
Weight	50	50	30	80	50	20	0
Profit	$340	$83.33	$180	$120	$220	$300	$0

Table 5: Result from Greedy Algorithm

6 Conclusion and Future Work

In this paper, we introduced dynamic programming, approximation, and greedy algorithms for the 0–1 knapsack problem and the fractional knapsack problem. We proved optimality of the greedy algorithm for the fractional case and established a 1/2-approximation guarantee for a modified greedy algorithm in the 0–1 case, while dynamic programming provided an exact baseline. We also illustrated how these methods can be applied in real-life practice. These algorithms help us make better and more rational decisions.

For future work, we plan to extend the empirical analysis to the multidimensional knapsack problem and explore additional algorithms (e.g., genetic algorithms) for knapsack problems. In addition, we may conduct a comparative analysis of the performance of different algorithms.

Appendix

Item	Discounted_Price	Original_Price	Discount
Formal shoe	$23	$36.99	$13
Tablet	$30	$49.99	$20
Headphone	$60	$149.99	$90
Printer	$20	$59.99	$40
Speaker	$45	$59.99	$15
Sneaker	$55	$89.95	$35
Running Shoe	$66	$85.00	$19
Watch	$75	$91.75	$17
Wireless speaker	$99	$199.00	$100
Jogger	$41	$54.00	$14
Arsenal kit	$84	$90.00	$6
Gym shoe	$57	$69.95	$13
Laptop bag	$98	$113.97	$16
Bluetooth headphone	$78	$129.99	$52
Skateboard	$37	$59.99	$23
Sweater	$26	$27.82	$2
PS controller	$39	$64.99	$26
GTA V game	$85	$99.99	$15
Bike helmet	$30	$33.96	$4
Hair dryer	$39	$59.99	$21
Bed sheet	$32	$39.78	$8
Shirt	$28	$39.99	$12
Power bank	$40	$46.99	$7
Sunglass	$35	$89.99	$55
Swimsuit	$32	$54.00	$22
Aquarium tank	$28	$32.89	$5
Blazer	$86	$99.99	$14
Kitchen mixer	$37	$39.67	$2
Microwave bowl	$16	$17.99	$2
Protein powder	$24	$28.99	$5

Table 6: Product Information For Setting 1

References

[1] Vassos Hadzilacos. Fractional knapsack, 2022.

[2] Bernhard Korte and Jens Vygen. *Combinatorial optimization: Theory and algorithms.* Springer Berlin Heidelberg, 2000.

[3] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and computer implementations.* UMI Books on Demand, 2006.

[4] Hayat Abdullah Shamakhai. The 0-1 multiple knapsack problem, 2017.

[5] Ben Hamida Yazid. Quantum-inspired genetic algorithm for a class of combinatorial optimization. Master's thesis, M'Sila, Algeria, 2016.