Multiple-Choice Knapsack with Small Items

Jakub Pawlewicz
Institute of Informatics, University of Warsaw, Poland

Abstract

The Multiple-Choice Knapsack problem is a generalization of the Knapsack problem in which items are partitioned into disjoint classes and exactly one item must be selected from each class. Besides the standard dynamic program running in time $\mathcal{O}(nc)$, where n is the total number of items and c is the knapsack capacity, no other pseudopolynomial-time algorithm is currently known. We present an $\mathcal{O}(n + w_{\max}^4)$-time algorithm, where $w_{\max} = \max_i(w_{in_i} - w_{i1})$ is the maximum weight range within a class.

2012 ACM Subject Classification Theory of computation → Dynamic programming; Mathematics of computing → Discrete optimization

Keywords and phrases multiple-choice knapsack, knapsack problem, small items, pseudopolynomial algorithms, proximity bounds, subset sums

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.29

Funding Supported by the National Science Centre (NCN) grant no. 2022/47/D/ST6/02184.

1 Introduction

The classical 0-1 Knapsack problem asks to choose a subset of items, each with weight and profit, whose total weight does not exceed a capacity, while maximizing the total profit. It is a cornerstone of combinatorial optimization and a canonical NP-hard problem, motivating a broad toolkit ranging from pseudopolynomial-time dynamic programs to approximation schemes and fine-grained algorithmic results.

The Multiple-Choice Knapsack problem (MCKP) extends the 0-1 Knapsack problem by partitioning items into disjoint classes and requiring exactly one item from each class. It captures settings where each component must be chosen from a menu of alternatives.

A closely related variant is the Selective MCKP, where at most one item per class may be selected. It reduces to MCKP by adding a dummy item of weight and price 0 to each class, so we focus on the exact-choice formulation without loss of generality.

MCKP appears in applications such as resource allocation, project selection, and communication systems; see the survey by Szkaliczki [12] for a recent overview or the book by Kellerer et al. [9, Chapter 11] for comprehensive coverage of Multiple-Choice Knapsack and related problems. In a typical application, each class represents a decision that must be made exactly once (e.g., selecting an implementation of a module, a mode of operation for a user, or one project from a group of mutually exclusive alternatives), and each item encodes a trade-off between resource consumption (weight) and benefit (price). The knapsack capacity models the global resource budget. MCKP also arises in string algorithmics, e.g., in weighted pattern matching and consensus problems on weighted sequences and profiles [10].

Applications with small within-class ranges. Many communication and sensing problems lead to MCKP instances in which each class has only a few discrete operating points, so the within-class weight range $w_{in_i} - w_{i1}$ is naturally small. In Section 8 we discuss representative examples including bit allocation for crowdsourcing-based target tracking and discrete transmit-power control in Zigbee sensor networks [4, 5]. We also note that the same dynamic-programming machinery yields bounded-domain $(\max, +)$ convolution on consecutive ranges, which connects to deterministic network calculus [6, 2].

For the 0-1 Knapsack problem, classical pseudopolynomial-time dynamic programs run in $\mathcal{O}(nc)$ for integral capacity c, or in $\mathcal{O}(nP)$ when parameterized by the total price P, and yield approximation schemes. In the small-weight regime, algorithms with running time $\tilde{\mathcal{O}}(n + w_{\max}^2)$ have been presented for 0-1 Knapsack [8, 3]. For MCKP, the standard $\mathcal{O}(nc)$ dynamic program remains essentially the only pseudopolynomial-time algorithm currently known; see, e.g., [7].

Our main result is the following.

▶ **Theorem 1.** *The Multiple-Choice Knapsack problem can be solved in time $\mathcal{O}(n + w_{\max}^4)$, where n is the total number of items and $w_{\max}$ is the maximum weight range within a class. Moreover, for any capacity c, the algorithm can output optimal objective values for the capacities $c, c+1, \ldots, c+w_{\max}$ within the same running time.*

When item prices are integral and the maximum within-class price range $p_{\max}$ is small, we also obtain a symmetric price-based variant running in $\mathcal{O}(n + p_{\max}^4)$ (Section 7).

At a high level, we first compute the upper hull of each class and use it to construct a greedy anchor (Section 4). We then prove a proximity lemma (Lemma 7) showing that there exists an optimal solution for capacity c that differs from the anchor in only few classes, with bounded total weight increase and decrease. The proof combines an exchange argument with an elementary subset-sum separation lemma for two multisets whose positive subset sums are disjoint (Section 5). Finally, we keep only the most promising candidates for each possible weight change and run a dynamic program over the total weight change within a window of size $\mathcal{O}(w_{\max}^2)$ (Section 6).

While MCKP is closely related to 0-1 Knapsack, standard reductions do not preserve the structural regimes exploited by the fastest "small item" algorithms. In particular, normalizing by the lightest item in each class creates upgrade items but also introduces a per-class restriction (at most one upgrade per class), which breaks the unrestricted mixing of items that those methods rely on. Our algorithm instead uses the greedy anchor as a reference point and applies a proximity bound to search only within a bounded neighborhood whose size is controlled by $w_{\max}$.

2 Preliminaries

We write $[k] = \{1, \ldots, k\}$ for $k \in \mathbb{N}$. A *multiset* is a set with multiplicities; $A' \subseteq A$ denotes a submultiset and $|A|$ is its size counting multiplicities. For a multiset A of integers we define $\Sigma(A)$ as the sum of elements and $\max(A)$ as the maximum element. We also use

$$\mathcal{S}(A) = \{\Sigma(A') \ : \ A' \subseteq A\}, \qquad \mathcal{S}^*(A) = \mathcal{S}(A) \setminus \{0\}.$$

3 Problem model

The Multiple-Choice Knapsack problem is a generalization of the Knapsack problem. Items are partitioned into classes, and exactly one item must be selected from each class.

3.1 Problem statement, notation, and assumptions

Formally, the input consists of

- the knapsack capacity: $c \in \mathbb{R}$,
- the number of classes: $r \in \mathbb{N}$,
- the number of items in each class: $n_i \in \mathbb{N}$ for $i \in [r]$,
- the weight and price of each item: $(w_{ij}, p_{ij}) \in \mathbb{R}$ for $i \in [r]$, $j \in [n_i]$.

A solution is an r-element sequence of indices $(j_1, \ldots, j_r)$, denoted by $\boldsymbol{j} \in J$, where $J = [n_1] \times \ldots \times [n_r]$ is the solution space. The weight and price of $\boldsymbol{j}$ are

$$w(\boldsymbol{j}) = \sum_{i=1}^{r} w_{ij_i}, \qquad p(\boldsymbol{j}) = \sum_{i=1}^{r} p_{ij_i}.$$

The goal is to maximize $p(\boldsymbol{j})$ subject to $w(\boldsymbol{j}) \le c$ over all $\boldsymbol{j} \in J$. If $\boldsymbol{j}$ maximizes $p(\boldsymbol{j})$ under the constraint $w(\boldsymbol{j}) \le c$, we say that $\boldsymbol{j}$ is an *optimal solution for capacity* c.

Each item is represented by a pair of indices (i, j), $i \in [r]$, $j \in [n_i]$. We denote the set of all items by $S = \{(i, j) : i \in [r], j \in [n_i]\}$ and the i-th class by $S_i = \{(i, j) : j \in [n_i]\}$. The items selected by a solution $\boldsymbol{j}$ are $S[\boldsymbol{j}] = \{(i, j_i) : i \in [r]\}$.

An item (w, p) is dominated within its class if there exists an item (w', p') in the same class with $w' \le w$ and $p' \ge p$. We assume that such items are removed. Therefore, within each class items are arranged in strictly increasing order of both weights and prices: $w_{i1} < \cdots < w_{in_i}$ and $p_{i1} < \cdots < p_{in_i}$. We assume that $\sum_{i=1}^{r} w_{i1} \le c$, since otherwise the instance is infeasible, and that $c < \sum_{i=1}^{r} w_{in_i}$, since otherwise the heaviest item in every class is optimal. Finally, item weights are assumed to be integers. We denote $w_{\max} = \max_{i=1}^{r}(w_{in_i} - w_{i1})$ and $n = \sum_{i=1}^{r} n_i$.

3.2 Relation to Selective Multiple-Choice Knapsack

The selective version (Selective MCKP, SMCKP) allows picking at most one item from each class, and classes may be skipped. It reduces to MCKP by adding to every class a dummy item of weight and price 0; picking this item corresponds to skipping the class. Conversely, MCKP is a special case of the selective version where choosing an item from every class is required. Hence any algorithm for one variant applies to the other with this simple reduction.

We can also reduce MCKP to SMCKP by normalizing each class with its lightest item. Let (w_{i1}, p_{i1}) be the lightest item in class i and set $c' = c - \sum_{i=1}^{r} w_{i1}$. For each $j > 1$ define an upgrade item with weight $w_{ij} - w_{i1}$ and price $p_{ij} - p_{i1}$. Selecting at most one upgrade per class in the selective instance corresponds to choosing exactly one item per class in the original instance, with the baseline items providing the option of no upgrade.

4 Greedy anchors

We start by defining the greedy anchor used as the reference point for the local search. The anchor is computed from the upper hull of every class, but the final dynamic program in Section 6 still considers the original items. This distinction is important: points below the upper hull need not be removed from the instance, but they are dominated by the piecewise-linear upper envelope used in the proof of optimality of the anchor.

▶ **Definition 2.** *For a class represented by upper-hull vertices, and for every $j \ge 2$, define the weight increment, price increment, and slope of the upgrade from item $j-1$ to item j by*

$$\Delta^w_{i,j} = w_{ij} - w_{i(j-1)}, \qquad \Delta^p_{i,j} = p_{ij} - p_{i(j-1)}, \qquad \rho_{i,j} = \frac{\Delta^p_{i,j}}{\Delta^w_{i,j}}.$$

An instance is concave *if $\rho_{i,j} \ge \rho_{i,j+1}$ for all $i \in [r]$ and $j \in \{2, \ldots, n_i - 1\}$, i.e., marginal slopes within each class are non-increasing.*

For each class of an arbitrary instance, we compute the vertices of the upper convex hull of the points (w_{ij}, p_{ij}). The resulting piecewise-linear envelope is concave as a function of weight; this is why we use the term "concave" for the corresponding upper-hull subinstance.

Since items in a class are already sorted by weight, the upper hull of class i can be computed in $\mathcal{O}(n_i)$ time by the standard monotone stack procedure, and hence all upper hulls can be computed in $\mathcal{O}(n)$ time.

The greedy procedure starts with the lightest upper-hull item in each class. It then processes upper-hull upgrades in non-increasing order of slope. Once an upgrade of the current maximum remaining slope does not fit into the remaining budget, the procedure may still take other upgrades of the same slope that fit, but it stops before taking any upgrade of smaller slope. This is the intended prefix-by-slope rule; in particular, the greedy anchor is not obtained by skipping an overweight high-slope upgrade and then continuing with lower-slope upgrades.

Let $w_0 = \sum_{i=1}^{r} w_{i1}$ and $B = c - w_0$ be the upgrade budget. A linear-time weighted selection on the slopes [1] finds a threshold τ such that all upgrades with slope larger than τ fit, while not all upgrades with slope at least τ fit. We then scan the classes once to take all larger-slope upgrades, and once more to take threshold-slope upgrades while they fit. The resulting solution is the *greedy anchor* for capacity c.

Algorithm 1 Greedy anchor for an upper-hull instance.

Require: Upper-hull classes $S_1, \dots, S_r$, capacity c
$j_i \leftarrow 1$ for all i, $w_0 \leftarrow \sum_{i=1}^{r} w_{i1}$
if $w_0 \geq c$ **then**
  return $\boldsymbol{g} = (j_1, \dots, j_r)$
end if
$B \leftarrow c - w_0$
Find a threshold τ such that $\sum_{\rho_{i,j} > \tau} \Delta^w_{i,j} \leq B < \sum_{\rho_{i,j} \geq \tau} \Delta^w_{i,j}$
for each class i **do**
  while $j_i + 1 \leq n_i$ and $\rho_{i,j_i+1} > \tau$ **do**
    $j_i \leftarrow j_i + 1$, $B \leftarrow B - \Delta^w_{i,j_i}$
  end while
end for
for each class i **do**
  while $j_i + 1 \leq n_i$ and $\rho_{i,j_i+1} = \tau$ and $\Delta^w_{i,j_i+1} \leq B$ **do**
    $j_i \leftarrow j_i + 1$, $B \leftarrow B - \Delta^w_{i,j_i}$
  end while
end for
return $\boldsymbol{g} = (j_1, \dots, j_r)$

▶ **Proposition 3** (Greedy anchor). *The greedy anchor can be computed in $\mathcal{O}(n)$ time. If Algorithm 1 returns $\boldsymbol{g}$ and $c_g = w(\boldsymbol{g})$, then $\boldsymbol{g}$ is an optimal solution for capacity c_g in the original instance. Moreover, $c - c_g < w_{\max}$.*

Proof. The running-time bound follows from the linear-time upper-hull computation, the linear-time weighted selection of the threshold slope, and the two scans over the upper-hull upgrades.

For optimality, consider for every class i the piecewise-linear upper envelope f_i obtained by interpolating between consecutive upper-hull vertices. Every original item (i, j) satisfies $p_{ij} \leq f_i(w_{ij})$, with equality for upper-hull vertices. Hence any feasible MCKP solution of weight at most c_g is feasible for the relaxation

$$\max \sum_{i=1}^{r} f_i(x_i) \quad \text{subject to} \quad \sum_{i=1}^{r} x_i \le c_g, \quad w_{i1} \le x_i \le w_{in_i},$$

and its price is at most the relaxation value.

The functions f_i are concave, and their linear pieces have slopes $\rho_{i,j}$. A standard exchange argument proves that a prefix of pieces in non-increasing slope order is optimal for the relaxation: if an allocation uses a positive amount of a lower-slope piece while a higher-slope piece is not full, shifting a sufficiently small amount of weight from the former to the latter preserves feasibility and does not decrease the objective (and increases it for a strict slope inequality). Algorithm 1 produces such a prefix at total weight c_g, with arbitrary tie-breaking among pieces of slope τ. Therefore the relaxation optimum at capacity c_g equals $p(\boldsymbol{g})$. Since $\boldsymbol{g}$ itself is an original feasible solution, it is optimal for the original instance at capacity c_g.

Finally, after the second scan no remaining upgrade of slope τ fits into the residual budget, and all lower-slope upgrades are deliberately ignored. The threshold exists because the all-heaviest solution is assumed to exceed c. Thus the residual budget $c - c_g$ is smaller than the weight of some remaining upper-hull upgrade, and every such upgrade has weight at most $w_{\max}$. ◀

5 A subset-sum separation lemma

To control how far an optimal solution can deviate from the greedy anchor, we need a bound on cancellations: when some classes switch to lighter items and others to heavier items, the corresponding weight differences form two multisets. If these multisets shared a nonzero common subset sum, we could exchange those classes and contradict the minimality argument in the proximity proof. The following elementary pigeonhole lemma is the same kind of subset-sum separation used by Polak, Rohwedder, and Węgrzycki for 0-1 Knapsack [11, Lemma 2.1]; we include a self-contained variant for the notation used here. Throughout this section, all multisets consist of positive integers.

▶ **Theorem 4.** *Assume we have two multisets A and B of positive integers with $\Sigma(A) = \Sigma(B) = s$ and $\mathcal{S}^*(A) \cap \mathcal{S}^*(B) = \{s\}$. Then $|A| \le \max(B)$ and $|B| \le \max(A)$. Thus for w such that $\max(A), \max(B) \le w$, we have $s \le w(w-1)$.*

Proof. Let $b_{\max} = \max(B)$, $n = |A|$, $m = |B|$. Enumerate all elements of A and B in no particular order: $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_m\}$. Define prefix sums $P_A(i) = \sum_{k \le i} a_k$ and $P_B(j) = \sum_{k \le j} b_k$ for $0 \le i \le n$ and $0 \le j \le m$. For each i with $1 \le i < n$, define $j^*(i) = \max\{j : P_B(j) \le P_A(i)\}$.

Observe that for $1 \le i < n$ we have $j^*(i) < m$, hence $b_{j^*(i)+1}$ is defined and $0 \le \Delta(i) = P_A(i) - P_B(j^*(i)) < b_{j^*(i)+1} \le b_{\max}$. If $\Delta(i) = 0$ then $P_A(i) = P_B(j^*(i))$ gives a nontrivial common sum, contradicting $\mathcal{S}^*(A) \cap \mathcal{S}^*(B) = \{s\}$. Therefore $\Delta(i) \in [1, b_{\max} - 1]$.

If $\Delta(i_1) = \Delta(i_2)$ for some $1 \le i_1 < i_2 < n$, then $P_A(i_2) - P_A(i_1) = P_B(j^*(i_2)) - P_B(j^*(i_1))$, which again contradicts the assumption $\mathcal{S}^*(A) \cap \mathcal{S}^*(B) = \{s\}$. Therefore the values $\Delta(i)$ are distinct for all $1 \le i < n$, which implies $n - 1 \le b_{\max} - 1$ and proves $|A| \le \max(B)$. By symmetry we obtain $|B| \le \max(A)$.

Now assume $\max(A), \max(B) \le w$. If $w \in B$ then $\max(A) < w$ and A contains at most w elements of value at most $w - 1$. If $w \notin B$ then A contains at most $w - 1$ elements of value at most w. Either way $s = \Sigma(A) \le w(w-1)$. ◀

▶ **Corollary 5.** *Assume we have two multisets A and B of positive integers with $\Sigma(B)-\Sigma(A) = d > 0$ such that for some w we have $\max(A), \max(B), d \leq w$. If $\mathcal{S}^*(A) \cap \mathcal{S}^*(B) = \emptyset$, then $|A|+1, |B| \leq w$ and $\Sigma(B) = \Sigma(A) + d \leq w(w-1)$.*

Proof. Apply Theorem 4 to $A \cup \{d\}$ and B. Put $s = \Sigma(A) + d = \Sigma(B)$. We only need to verify that the positive subset sums of $A \cup \{d\}$ and B intersect only at s.

Suppose $\Sigma(A') = \Sigma(B')$ for some $A' \subseteq A \cup \{d\}$ and $B' \subseteq B$. If $d \notin A'$, then the common sum lies in $\mathcal{S}^*(A) \cap \mathcal{S}^*(B)$ unless it is zero; hence $A' = B' = \emptyset$.

Now let $d \in A'$. Consider the complements $\bar{A} = A \cup \{d\} \setminus A'$ and $\bar{B} = B \setminus B'$. Since $\Sigma(A) + d = \Sigma(B)$ and $\Sigma(A') = \Sigma(B')$, we have $\Sigma(\bar{A}) = \Sigma(\bar{B})$. Moreover $\bar{A} \subseteq A$ and $\bar{B} \subseteq B$. If this common sum were positive, it would belong to $\mathcal{S}^*(A) \cap \mathcal{S}^*(B)$, impossible. Thus $\bar{A} = \bar{B} = \emptyset$, and so $A' = A \cup \{d\}$ and $B' = B$. ◀

6 Proximity

Intuitively, the greedy anchor $\boldsymbol{g}$ is already close to optimal for capacity c: it may leave some unused slack, but the slack is bounded by $w_{\max}$. Any optimal solution must therefore differ from $\boldsymbol{g}$ in only a small number of classes, and large cancellations between weight decreases and increases would allow an exchange that improves $\boldsymbol{g}$. We turn this observation into an algorithm in three steps. First, we show that some optimal solution lies within a bounded neighborhood of the anchor. Next, we retain only $\mathcal{O}(w_{\max}^2)$ candidate changes. Finally, we solve the resulting reduced instance by dynamic programming.

6.1 Optimal solutions near an anchor

We begin by introducing notation for the differences between two solutions.

▶ **Definition 6.** *For two solutions $\boldsymbol{g}, \boldsymbol{j} \in J$ we define the following sets of class indices representing the differences between these solutions: $D^\bullet(\boldsymbol{g}, \boldsymbol{j}) = \{i \in [r] \,:\, w_{ig_i} \bullet w_{ij_i}\}$ where $\bullet$ is a comparison relation: $\bullet \in \{\neq, >, <\}$. Thus $D^{\neq}(\boldsymbol{g}, \boldsymbol{j})$ denotes the set of class indices where solutions differ, $D^{<}(\boldsymbol{g}, \boldsymbol{j})$ those where the second solution has a heavier item, and $D^{>}(\boldsymbol{g}, \boldsymbol{j})$ those where the second solution has a lighter item.*

Additionally, we define the multiset of weight differences $D^\bullet_w(\boldsymbol{g}, \boldsymbol{j}) = \{|w_{ij_i} - w_{ig_i}| \,:\, i \in D^\bullet(\boldsymbol{g}, \boldsymbol{j})\}$ for $\bullet \in \{>, <\}$. The multiset $D^{\geq}_w(\boldsymbol{g}, \boldsymbol{j})$ (respectively, $D^{\leq}_w(\boldsymbol{g}, \boldsymbol{j})$) contains the weight differences by which the item weight was decreased (increased) between solutions $\boldsymbol{g}$ and $\boldsymbol{j}$.

▶ **Lemma 7.** *Let $\boldsymbol{g}$ be an optimal solution for capacity c' with $w(\boldsymbol{g}) = c'$ and $0 \leq c - c' \leq w_{\max}$. Then there exists an optimal solution $\boldsymbol{j}$ for capacity c such that*

- $|D^{>}(\boldsymbol{g}, \boldsymbol{j})| < w_{\max}$ *and* $\Sigma(D^{\geq}_w(\boldsymbol{g}, \boldsymbol{j})) < w_{\max}(w_{\max} - 1)$,
- $|D^{<}(\boldsymbol{g}, \boldsymbol{j})| \leq w_{\max}$ *and* $\Sigma(D^{\leq}_w(\boldsymbol{g}, \boldsymbol{j})) \leq w_{\max}(w_{\max} - 1)$.

Proof. Let $\boldsymbol{j}$ be an optimal solution for capacity c that minimizes the size of $D^{\neq}(\boldsymbol{g}, \boldsymbol{j})$. If $w(\boldsymbol{j}) \leq c'$, then $\boldsymbol{j}$ is feasible for capacity c'. Since $\boldsymbol{g}$ is optimal for capacity c', we have $p(\boldsymbol{g}) \geq p(\boldsymbol{j})$. But $\boldsymbol{g}$ is feasible for capacity c and $\boldsymbol{j}$ is optimal for capacity c, so $\boldsymbol{g}$ is also optimal for capacity c. Replacing $\boldsymbol{j}$ by $\boldsymbol{g}$ gives empty difference sets, which satisfy the stated bounds. Thus assume from now on that $w(\boldsymbol{j}) > c'$.

Let $d = w(\boldsymbol{j}) - w(\boldsymbol{g})$. Then

$$\Sigma(D^{\geq}_w(\boldsymbol{g}, \boldsymbol{j})) + d = \Sigma(D^{\leq}_w(\boldsymbol{g}, \boldsymbol{j})),$$

and $0 < d \leq c - c' \leq w_{\max}$.

We claim that

$$\mathcal{S}^*(D^{>}_w(\boldsymbol{g},\boldsymbol{j})) \cap \mathcal{S}^*(D^{<}_w(\boldsymbol{g},\boldsymbol{j})) = \emptyset.$$

Suppose otherwise. Then there exist nonempty sets $I^{>} \subseteq D^{>}(\boldsymbol{g},\boldsymbol{j})$ and $I^{<} \subseteq D^{<}(\boldsymbol{g},\boldsymbol{j})$ such that

$$\sum_{i\in I^{>}} (w_{ig_i} - w_{ij_i}) = \sum_{i\in I^{<}} (w_{ij_i} - w_{ig_i}).$$

Let $I = I^{>} \cup I^{<}$. Then $\sum_{i\in I} w_{ig_i} = \sum_{i\in I} w_{ij_i}$. Since $\boldsymbol{g}$ is optimal for capacity c' in the original instance, we must have $\sum_{i\in I} p_{ig_i} \geq \sum_{i\in I} p_{ij_i}$; otherwise swapping the classes of I in $\boldsymbol{g}$ to the items of $\boldsymbol{j}$ would preserve weight and strictly increase price.

Now form $\boldsymbol{j}^*$ from $\boldsymbol{j}$ by replacing, for every $i \in I$, the item of $\boldsymbol{j}$ by the item of $\boldsymbol{g}$. The total weight is preserved and the price does not decrease: $w(\boldsymbol{j}^*) = w(\boldsymbol{j})$ and $p(\boldsymbol{j}^*) \geq p(\boldsymbol{j})$. Hence $\boldsymbol{j}^*$ is also optimal for capacity c, but it differs from $\boldsymbol{g}$ in fewer classes than $\boldsymbol{j}$ does, contradicting the choice of $\boldsymbol{j}$.

Therefore the multisets $A = D^{\geq}_w(\boldsymbol{g},\boldsymbol{j})$ and $B = D^{\leq}_w(\boldsymbol{g},\boldsymbol{j})$ satisfy the assumptions of Corollary 5 with $w = w_{\max}$. The stated bounds follow. ◀

6.2 Candidate pruning

Given an anchor $\boldsymbol{g}$, exchanging the selected item in class i with item (i,j) changes the weight and price by

$$\omega_{ij} = w_{ij} - w_{ig_i}, \qquad \varphi_{ij} = p_{ij} - p_{ig_i}.$$

The notation $\omega_{ij}, \varphi_{ij}$ is used only for changes relative to $\boldsymbol{g}$; it is distinct from the upper-hull increments $\Delta^w_{i,j}, \Delta^p_{i,j}$ used in Section 4. For every nonzero integer

$$\delta \in \{-w_{\max}, \ldots, -1, 1, \ldots, w_{\max}\}, \qquad G_\delta = \{(i,j) : \omega_{ij} = \delta\}.$$

We retain, in each group G_δ, the $2w_{\max}$ items with largest price change φ_{ij} (or all items if the group is smaller). Let $S_{\boldsymbol{g}}$ be the union of retained items. Then $|S_{\boldsymbol{g}}| \leq 4w^2_{\max}$.

The factor $2w_{\max}$ is chosen to dominate the total proximity bound $2w_{\max} - 1$, not the number of changes in one weight-change group. This extra slack is needed because, when repairing a fixed group G_δ, some of its most profitable retained candidates may lie in classes already changed through other groups. Such candidates are unusable under the "one item per class" constraint.

▶ **Lemma 8** (Candidate pruning). *Among the optimal solutions for capacity c guaranteed by Lemma 7, there is one whose every changed item belongs to $S_{\boldsymbol{g}}$.*

Proof. Choose, among the optimal solutions satisfying Lemma 7, a solution $\boldsymbol{j}$ that maximizes the number of changed classes whose corresponding item is retained. Put $T = |D^{\neq}(\boldsymbol{g},\boldsymbol{j})|$. By Lemma 7, $T \leq 2w_{\max} - 1$.

Suppose that, for some group G_δ, the solution $\boldsymbol{j}$ uses an unretained item. Let q be the number of changed classes of $\boldsymbol{j}$ in G_δ. Since weights are strictly ordered inside every class, each class contributes at most one candidate to a fixed group G_δ. Hence each of the $T - q$ classes changed by $\boldsymbol{j}$ in a different group can forbid at most one retained candidate of G_δ. Among the $2w_{\max}$ retained candidates of G_δ, at least $2w_{\max} - (T - q) > q$ are therefore not forbidden. Choose q such retained candidates with largest price changes. Every retained

candidate has price change at least that of every unretained candidate in the same group, so the total price change of the chosen retained candidates is at least the total price change of the q items used by $\boldsymbol{j}$ in G_δ.

Replacing the G_δ changes of $\boldsymbol{j}$ by these chosen retained changes preserves class feasibility, preserves the total weight change because all involved items have the same $\omega_{ij} = \delta$, and does not decrease the price. The proximity bounds are preserved as well. The resulting solution is optimal and has more retained changed items than $\boldsymbol{j}$, a contradiction. ◀

6.3 Dynamic programming around an anchor

Let $W = w_{\max}(w_{\max} - 1)$. For each class i define

$$\mathcal{C}_i = \{(0,0)\} \cup \{(\omega_{ij}, \varphi_{ij}) : (i,j) \in S_{\boldsymbol{g}}\}.$$

Only classes with $\mathcal{C}_i \neq \{(0,0)\}$ are processed. For these classes we compute a rolling dynamic program over total weight change $x \in [-W, W]$. After processing some classes, $\mathrm{DP}(x)$ stores the maximum total price change achievable with total weight change x; unreachable states have value $-\infty$. The transition for a processed class i is

$$\mathrm{DP}'(x+\omega) = \max\{\mathrm{DP}'(x+\omega), \mathrm{DP}(x) + \varphi\} \qquad \text{for } (\omega, \varphi) \in \mathcal{C}_i.$$

Algorithm 2 Dynamic program around an anchor solution.

Require: Classes $S_1, \ldots, S_r$, anchor solution $\boldsymbol{g}$, capacity c
$W \leftarrow w_{\max}(w_{\max} - 1)$, $B \leftarrow c - w(\boldsymbol{g})$
For each nonzero $\delta \in [-w_{\max}, w_{\max}]$, keep the $2w_{\max}$ items of G_δ with largest φ_{ij}
For each class i build $\mathcal{C}_i = \{(0,0)\} \cup \{(\omega_{ij}, \varphi_{ij}) : (i,j) \in S_{\boldsymbol{g}}\}$
Initialize $\mathrm{DP}(x) = -\infty$ for $x \in [-W, W]$ and set $\mathrm{DP}(0) = 0$
for each class i with $\mathcal{C}_i \neq \{(0,0)\}$ **do**
    Initialize $\mathrm{DP}'(x) = -\infty$ for all $x \in [-W, W]$
    for each $x \in [-W, W]$ **do**
        for each $(\omega, \varphi) \in \mathcal{C}_i$ **do**
            if $x + \omega \in [-W, W]$ **then**
                $\mathrm{DP}'(x+\omega) \leftarrow \max\{\mathrm{DP}'(x+\omega), \mathrm{DP}(x) + \varphi\}$
            end if
        end for
    end for
    $\mathrm{DP} \leftarrow \mathrm{DP}'$
end for
return $p(\boldsymbol{g}) + \max\{\mathrm{DP}(x) : x \in [-W, W],\ x \leq B\}$

▶ **Proposition 9** (DP correctness and running time). *Let $\boldsymbol{g}$ be an optimal solution for capacity $c' = w(\boldsymbol{g})$ with $0 \leq c - c' \leq w_{\max}$. Algorithm 2 outputs the optimal value for capacity c in the original instance. Its running time is $\mathcal{O}(n + w_{\max}^4)$ and its working space is $\mathcal{O}(w_{\max}^2)$.*

Proof. By Lemma 7 and Lemma 8, there exists an optimal solution for capacity c whose changed items are all in $S_{\boldsymbol{g}}$ and whose total weight change lies in $[-W, W]$. The dynamic program enumerates exactly all choices of at most one retained change per class, grouped by their total weight change. Therefore the final maximum over $x \leq c - w(\boldsymbol{g})$ contains the value of an optimal solution and cannot exceed the optimum, since every DP state corresponds to a feasible set of class changes.

The pruning step is implemented by linear-time selection in each weight change group, and the groups together contain $O(n)$ items. The number of retained candidates is $M = |S_{\boldsymbol{g}}| \le 4w_{\max}^2$, so the number of active classes is at most M. The DP uses two arrays indexed by $x \in [-W, W]$, hence $\mathcal{O}(W) = \mathcal{O}(w_{\max}^2)$ space. Processing a class i costs $\mathcal{O}(W|\mathcal{C}_i|)$ time, and over all active classes $\sum_i |\mathcal{C}_i| \le 2M$. Thus the DP time is $\mathcal{O}(WM) = \mathcal{O}(w_{\max}^4)$, and the total time is $\mathcal{O}(n + w_{\max}^4)$. ◀

▶ **Corollary 10** (Consecutive capacities). *Fix an anchor $\boldsymbol{h}$ that is optimal for capacity $c_h = w(\boldsymbol{h})$. One run of the dynamic program around $\boldsymbol{h}$ gives optimal values for every capacity $\hat{c} \in \{c_h, c_h + 1, \ldots, c_h + w_{\max}\}$.*

Proof. The DP table computed around $\boldsymbol{h}$ is independent of the final residual capacity. By Proposition 9, for every such $\hat{c}$ the optimal value is

$$p(\boldsymbol{h}) + \max\{\mathrm{DP}_{\boldsymbol{h}}(x) : x \in [-W, W],\ x \le \hat{c} - c_h\}.$$

After computing prefix maxima of $\mathrm{DP}_{\boldsymbol{h}}(x)$ over the ordered range $x = -W, -W+1, \ldots, W$, all $w_{\max} + 1$ values are read off in $\mathcal{O}(w_{\max})$ additional time. ◀

We can now prove Theorem 1. Compute the upper hulls and the greedy anchor $\boldsymbol{g}$ for the requested capacity c. By Proposition 3, $\boldsymbol{g}$ is optimal for $c_g = w(\boldsymbol{g})$ in the original instance and $c - c_g < w_{\max}$. Therefore Proposition 9 gives the optimal value for capacity c.

For the consecutive-capacity statement, Corollary 10 applied to $\boldsymbol{g}$ gives optimal values for $c_g, c_g + 1, \ldots, c_g + w_{\max}$. If these already cover $c, c+1, \ldots, c+w_{\max}$, we are done. Otherwise, continue the greedy sweep by one more upper-hull upgrade and let $\boldsymbol{g}'$ be the resulting anchor. The next upgrade has weight at most $w_{\max}$, hence

$$c < w(\boldsymbol{g}') \le w(\boldsymbol{g}) + w_{\max}.$$

Corollary 10 applied to $\boldsymbol{g}'$ gives optimal values for $w(\boldsymbol{g}'), w(\boldsymbol{g}') + 1, \ldots, w(\boldsymbol{g}') + w_{\max}$. The two intervals cover all capacities $c, c+1, \ldots, c+w_{\max}$. Running the dynamic program once or twice changes the running time only by a constant factor, so the total time remains $\mathcal{O}(n + w_{\max}^4)$.

7 Small item prices

Assume prices are integers and define $p_{\max} = \max_{i \in [r]}(p_{in_i} - p_{i1})$. The price-based variant mirrors the weight-based approach, but with roles reversed. First, the greedy anchor is constructed by following the steepest price-per-weight slopes, and it leaves a bounded *price slack* rather than a weight slack. That is, once the greedy process stops, any remaining upgrade would exceed the capacity, so the additional price that could be gained by further upgrades is limited by the largest price step in any class. Second, we group candidate upgrades by their price change Δp (instead of weight change) and keep, for each Δp, the $2p_{\max}$ items with the smallest weight increase. This guarantees that the multiset of price changes used by an optimal solution has total price change at most $P = p_{\max}(p_{\max} - 1)$.

We then run a dynamic program indexed by total price change $y \in [-P, P]$, where the state stores the minimum weight increase achievable with those upgrades. This is the dual of the weight-based DP, which stores maximum price increase for a given weight change. The same counting argument gives $\mathcal{O}(p_{\max}^2)$ candidates and total running time $\mathcal{O}(n + p_{\max}^4)$.

8 Applications

The running time of the standard dynamic program for MCKP is $\mathcal{O}(nc)$ for integral capacity c, which can be prohibitive when the global budget is large. Our bound $\mathcal{O}(n + w_{\max}^4)$ replaces this dependence on the global budget by $w_{\max} = \max_i(w_{in_i} - w_{i1})$, the largest *within-class* weight range. This regime is natural in engineered systems where each decision offers only a few discrete operating points. The same bounded-window dynamic program can also be viewed as a routine for bounded-domain $(\max, +)$ convolution on consecutive output ranges.

8.1 Bounded-domain $(\max, +)$ convolution

Our algorithm can be interpreted as evaluating a bounded-domain $(\max, +)$ convolution. Given functions $f_1, \ldots, f_k : \{0, \ldots, d\} \to \mathbb{R}$, define

$$F(x) = \max_{x_1 + \cdots + x_k = x} \sum_{i=1}^{k} f_i(x_i).$$

A straightforward dynamic programming approach computes $F(0), \ldots, F(d)$ in $\mathcal{O}(kd^2)$ time. Our algorithm implies that values of $F(x)$ on $d+1$ consecutive points can be computed in time $\mathcal{O}(dk + d^4)$. Consequently, values on an interval of length s can be obtained in $\mathcal{O}(s(k + d^3))$ time. This improves over the classical $\mathcal{O}(skd)$ dynamic programming approach when $k \gg d^2$.

8.2 Deterministic network calculus

Deterministic network calculus models network elements by service curves and combines them using min-plus convolution. For servers with service curves $\beta_1, \ldots, \beta_k$, the end-to-end service curve is

$$(\beta_1 \otimes \cdots \otimes \beta_k)(t) = \inf_{t_1 + \cdots + t_k = t} \sum_{i=1}^{k} \beta_i(t_i).$$

Such convolutions are a central computational primitive in DNC analyses of delay and backlog bounds [6, 2].

After a sign change this operation becomes a $(\max, +)$ convolution. When service curves are discretized on a bounded time horizon $t \in \{0, \ldots, d\}$, evaluating end-to-end guarantees reduces to computing such bounded-domain convolutions.

In many scenarios one is interested only in values of the convolution within a limited time window (for example when testing deadline feasibility or during parametric search for delay bounds). In such cases the windowed computation enabled by our algorithm can reduce the running time when the number of network elements is large but the discretization horizon is moderate.

8.3 Bit allocation under a global communication budget

Cao, Brahma, and Varshney study incentive-compatible target tracking via crowdsourcing under a global communication budget. In their formulation, the fusion center chooses for each sensor a number of bits to purchase subject to a total bit budget, which they cast as an MCKP and solve by dynamic programming in $\mathcal{O}(NM)$ time for N sensors and budget M [4]. When each sensor has only a small set of admissible bit depths (e.g., due to quantizer

resolution or protocol constraints), the per-sensor bit range is small, so $w_{\max}$ is bounded independently of M and the MCKP subproblem can be solved in time $\mathcal{O}(n + w_{\max}^4)$. This is particularly beneficial when the allocation must be recomputed repeatedly, e.g., across time steps in tracking or as part of a larger mechanism.

8.4 Transmit-power allocation with discrete power levels

Consolini, Medagliani, and Ferrari consider transmit-power allocation in Zigbee wireless sensor networks with a finite total power budget. They define a discrete optimization problem in which each node selects one transmit power from a small discrete set under a global power constraint, observe that it corresponds to the multiple-choice knapsack problem, and solve it using an off-the-shelf solver (MOSEK), noting that computation time grows quickly with the number of sensors [5]. Here the set of available power levels per node is typically small, so after discretization the within-class range $w_{\max}$ is small even if the total budget is large. In this setting, an $\mathcal{O}(n + w_{\max}^4)$-time algorithm gives an exact alternative whose running time is controlled by the discretization granularity rather than by the global budget.

9 Conclusions

We presented a pseudopolynomial-time algorithm for MCKP with running time $\mathcal{O}(n + w_{\max}^4)$, based on upper-hull greedy anchors, a proximity bound, and a subset-sum separation argument. The same dynamic program also yields optimal values for $w_{\max} + 1$ consecutive capacities. This windowed viewpoint leads to applications in bounded-domain $(\max, +)$ convolution, including deterministic network calculus. For instances with small price ranges, the same approach yields a symmetric bound of $\mathcal{O}(n + p_{\max}^4)$. An open problem is to match the best known small-item bounds for 0-1 Knapsack and improve the dependence on $w_{\max}$ or $p_{\max}$ for MCKP. For 0-1 Knapsack, stronger bounds in the small-item regime rely on refined additive-combinatorial tools that exploit unrestricted mixing of items. These techniques do not transfer cleanly to MCKP: the class constraint prevents arbitrary combinations and breaks the additive structure that those methods use. Closing this gap remains open. In particular, the $\mathcal{O}(n + w_{\max}^3)$ bound for 0-1 Knapsack from [11] relies on structural properties of unrestricted item sets and does not seem to extend to the class-constrained setting.

References

1 Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, August 1973. Funding Information: work was supported by the National Science Foundation under grant GJ-992. doi:10.1016/S0022-0000(73)80033-9.

2 Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001. doi:10.1007/3-540-45318-0.

3 Karl Bringmann. Knapsack with small items in near-quadratic time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC '24)*, 2024. doi:10.1145/3618260.3649719.

4 Nianxia Cao, Swastik Brahma, and Pramod K. Varshney. Target tracking via crowdsourcing: A mechanism design approach. *IEEE Transactions on Signal Processing*, 63(6):1464–1476, 2015. doi:10.1109/TSP.2015.2398838.

5 Luca Consolini, Paolo Medagliani, and Gianluigi Ferrari. Adjacency matrix-based transmit power allocation strategies in wireless sensor networks. *Sensors*, 9(7):5390–5422, 2009. doi:10.3390/s90705390.

6 René L. Cruz. A calculus for network delay, part i: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991. doi:10.1109/18.61109.

7 Krzysztof Dudziński and Stanisław Walukiewicz. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research*, 28(1):3–21, January 1987. doi:10.1016/0377-2217(87)90165-2.

8 Ce Jin. 0-1 knapsack in nearly quadratic time. In *Proceedings of the 56th Annual ACM Symposium on Theory of Computing (STOC '24)*, 2024. doi:10.1145/3618260.3649618.

9 Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-24777-7.

10 Tomasz Kociumaka, Solon P. Pissis, and Jakub Radoszewski. Pattern matching and consensus problems on weighted sequences and profiles. *Theory of Computing Systems*, 63(3):506–542, August 2018. doi:10.1007/s00224-018-9881-2.

11 Adam Polak, Lars Rohwedder, and Karol Węgrzycki. Knapsack and subset sum with small items. In *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *LIPIcs*, pages 106:1–106:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ICALP.2021.106.

12 Tibor Szkaliczki. Solution methods for the multiple-choice knapsack problem and their applications. *Mathematics*, 13(7), 2025. doi:10.3390/math13071097.