The colored knapsack problem: structural properties and exact algorithms

Fabio Ciccarelli[a], Alexander Helber[b], Erik Mühmer[b]

a Department of Computer Control and Management Engineering Antonio Ruberti Sapienza University of Rome Rome Italy

b Chair of Operations Research RWTH Aachen University Aachen Germany

Abstract

We introduce and study a novel generalization of the classical Knapsack Problem (KP), called the *Colored Knapsack Problem* (CKP). In this problem, the items are partitioned into classes of colors and the packed items need to be ordered such that no consecutive items are of the same color. We establish that the problem is weakly NP-hard and propose two exact dynamic programming algorithms with time complexities of $\mathcal{O}(bn^4)$ and $\mathcal{O}(b^2n^3)$, respectively. To enhance practical performance, we derive various dominance and fathoming rules for both approaches. From a theoretical perspective, we analyze the linear programming relaxation of the natural CKP formulation, proving that an optimal solution exists with at most two fractional items. We also show that the relaxation can be solved in $\mathcal{O}(n)$ time, matching the complexity of the classical KP. Finally, we establish a comprehensive benchmark of CKP instances, derived from the Colored Bin Packing Problem. Extensive computational experiments demonstrate that the proposed dynamic programming algorithms significantly outperform state-of-the-art MIP solvers on most of these instances.

Keywords: Combinatorial Optimization, Knapsack Problem, Integer Linear Programming, Dynamic Programming, Computational Experiments

1. Introduction

Given a set $\mathcal{I} = \{1, 2, \ldots, n\}$ of *n items*, each having an associated *profit* $p_i \in \mathbb{Z}_{>0}$ and *weight* $w_i \in \mathbb{Z}_{>0}$ $(i \in \mathcal{I})$, and a *capacity* $b \in \mathbb{Z}_{>0}, b < \sum_{i=1}^{n} w_i$, the *Knapsack Problem* (KP) asks to select a subset of items $S \subseteq \mathcal{I}$ which has maximum total profit, and a total weight not exceeding b. The classical *Integer Linear Programming* (ILP) formulation for the KP reads as follows:

$$(\text{ILP}_{\text{KP}}) \qquad \max \quad \sum_{i \in \mathcal{I}} p_i \, x_i \qquad \qquad (1a)$$

$$\sum_{i \in \mathcal{I}} w_i\, x_i \le b, \tag{1b}$$

$$x_i \in \{0,1\}, \qquad\qquad i \in \mathcal{I}, \tag{1c}$$

where x_i is a binary variable taking value 1 if and only if item $i \in \mathcal{I}$ is selected. The objective function (1a) equals the total profit of the selected items, while the *capacity constraint* (1b) ensures that the total weight of the selected items does not exceed the knapsack capacity. This problem has been the subject of intensive research during the last decades, see, e.g., the books by [9] and [11].

We study a variant of the KP in which we are also given a set of m colors $\mathcal{C} = \{1, 2, \ldots, m\}$, and each item $i \in \mathcal{I}$ has a color $\kappa_i \in \mathcal{C}$. The *Colored Knapsack Problem* (CKP) consists of the KP with the additional condition that there must exist an ordering of the selected items such that no two consecutive items share the same color. Borges et al. [3] showed that this is always possible as long as the number of items of each color in the knapsack is less than or equal to that of all other colors together, plus one. When each item is assigned a distinct color (i.e., there is no pair of distinct items $i, j \in \mathcal{I}$ such that $\kappa_i = \kappa_j$), the CKP reduces to the KP. Consequently, the CKP inherits the $\mathcal{NP}$-hardness of the KP; in particular, it is at least weakly $\mathcal{NP}$-hard. Note that, while for the KP it can be assumed that each item has positive profit, in the CKP items of zero or even negative profit may be packed to respect the color condition in an optimal solution. We therefore allow $\boldsymbol{p} \in \mathbb{Z}^n$ for instances of the CKP.

For each color $c \in \mathcal{C}$, let $\mathcal{I}_c := \{i \in \mathcal{I} \mid \kappa_i = c\}$ be the set of the items of color c and $\bar{\mathcal{I}}_c = \mathcal{I} \setminus \mathcal{I}_c$ the set of the items of all other colors. Then, the following is a natural ILP formulation for the CKP, denoted as $\mathrm{ILP}_{\mathrm{CKP}}$:

$$(\mathrm{ILP}_{\mathrm{CKP}}) \qquad \max\ \sum_{i \in \mathcal{I}} p_i\, x_i \tag{2a}$$

$$\sum_{i \in \mathcal{I}} w_i\, x_i \le b, \tag{2b}$$

$$\sum_{i \in \mathcal{I}_c} x_i - \sum_{i \in \bar{\mathcal{I}}_c} x_i \le 1, \qquad\qquad c \in \mathcal{C}, \tag{2c}$$

$$x_i \in \{0,1\}, \qquad\qquad i \in \mathcal{I}. \tag{2d}$$

The objective function (2a) and Constraint (2b) are identical to the objective function and the capacity constraint of $\mathrm{ILP}_{\mathrm{KP}}$, respectively. Constraints (2c), which we refer to as *color constraints*, are instead specific to the CKP and enforce that, for each color $c \in \mathcal{C}$, the number of items of color c in the knapsack cannot exceed the number of selected items of all other colors, plus one.

We now introduce some notation and terminology that will be useful in the remainder of the paper. For any subset of items $S \subseteq \mathcal{I}$, let $k_c(S) := |S \cap \mathcal{I}_c|$ be the number of items of color c in S. We call a color c *dominant* in S if it has the maximum cardinality among all colors, i.e., $c \in \arg\max_{c' \in \mathcal{C}}\{k_{c'}(S)\}$. Note that a set S may admit multiple dominant colors. A dominant color c is also *critical* for S if $k_c(S) = |S \setminus \mathcal{I}_c| + 1$. We observe that there can be at most one critical color in any subset of items $S \subseteq \mathcal{I}$. Indeed, for a color c to be critical, it must satisfy $k_c(S) > |S|/2$, meaning that more than half of the items in S share color c. Clearly, this condition cannot hold for more than one color.

The following example, illustrated in Figure 1, considers a CKP instance and compares an optimal KP solution, obtained by ignoring item colors, with an optimal CKP solution to that instance. It also shows that, even when a KP-optimal solution is infeasible for the CKP due to the violation of the color condition, there may not exist a CKP-optimal solution in which that color is critical, nor even dominant. Specifically, in the optimal KP solution (a), the subset of selected items is $S = \{1, 2\}$ (total profit 23). It is evident that S is not feasible for the CKP, since $k_1(S) > |S \setminus \mathcal{I}_1| + 1$. On the other hand, the optimal CKP solution (b) selects items $S = \{1, 3, 4\}$ (total profit 19). In this case, $c = 2$ is the critical color.
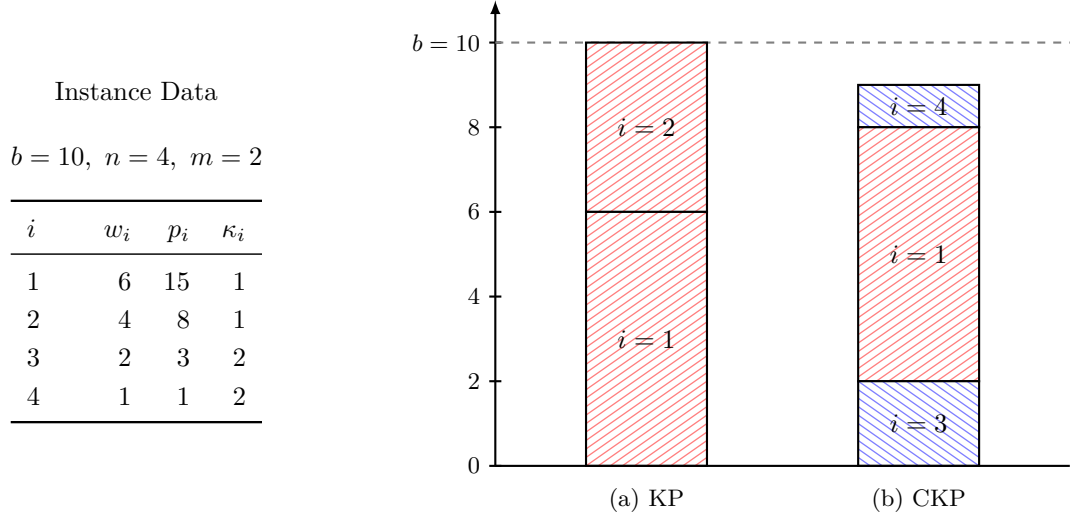

Figure 1: Example of a CKP instance. The instance parameters are detailed in the table to the left. The figure to the right shows an optimal KP solution (a), obtained by ignoring the item colors, and an optimal CKP solution (b) for the instance. The vertical axis shows the capacity consumption, and the height of each item corresponds to its weight. Items of color $c = 1$ are highlighted in red, while those of color $c = 2$ are blue.

1.1. Motivation and related problems

To the best of our knowledge, the CKP has not been studied yet. The closest problem to it, and the one that inspired this work, is the *Colored Bin Packing Problem* (CBPP). In this problem, items

3

of varying sizes and colors need to be packed in a minimal number of bins of a given size. For each bin, a color condition as in the CKP is enforced, i.e., the items packed in the bin need to be ordered so that no consecutive items are of the same color. If the CBPP is solved by branch-and-price, the subproblem is a CKP, motivating the study of this problem. So we refer to the literature on the CBPP for applications and motivation, except for one application mentioned by Balogh et al. [1] for which the CKP itself is also relevant: consider the programming of content, for example deciding which videos to show in the feed of a social media app on a phone screen. The content may belong to different categories (educational, funny, music, advertisements, etc.) or be from different content creators and advertisement customers, and a profitable set of videos should be selected to be shown to the user while not showing two videos of the same category consecutively.

The CBPP was first proposed for the two-color case by Balogh et al. [1], who investigated the online variant of the problem. They followed up with a study of the offline version of the problem, for which they developed approximation algorithms [2]. The problem was then generalized for the case with more than two colors by Dósa and Epstein [6], who also studied the online variant of the problem. Borges et al. [3] provides a summary of the results for the online variant and approximation algorithms for the offline variant.

Most relevant for our present work, Borges et al. [3] also propose various exact approaches for solving the offline variant of the CBPP. First, they introduce a compact (polynomial-size) binary programming model for the problem. Further, they propose two arc-flow models of pseudo-polynomial size, inspired by the model for the regular Bin Packing Problem of Valério De Carvalho [15], that they solve with a commercial solver. In these models, a solution is a flow through a network that represents a capacity-expansion of a bin, and arcs represent packing items. Each unit of flow represents the packing pattern for a single bin, retrievable by path-decomposition of the solution. The benefit of these models is twofold: first, they improve the quality of the linear programming relaxation bound significantly; second, they eliminate some symmetry inherent in the compact model by not explicitly modeling the assignment of items to specific bins. To enforce the color constraints, both models explicitly consider the ordering of the items. This makes these types of models unpromising for the knapsack variant, since the previously mentioned characterization that simply counts the number of items in each color is much more lightweight. They also introduce two exponential size set partitioning models for the CBPP, that they solve with Branch-and-Price using VRPSolver [14]. The pricing problems in those approaches are Colored Knapsack Problems, at least at the root node. They model them as Resource-Constrained Shortest Path Problems on two different graphs. The first variant uses

one resource per color, while the second one requires elementary paths to prevent items from being packing more than once. It is worth noticing that, in general, the Resource-Constrained Shortest Path Problem is strongly $\mathcal{NP}$-hard if the number of resources is part of the input (reduction from Binary Programming), or if the paths have to be elementary (reduction from Hamiltonian Path).

Apart from the bin packing variant, some of the plentiful knapsack variants are also similar to our problem. We refer to [4] for an overview of previously studied knapsack variants. Among the variants they mention, the most similar in spirit is perhaps the well-known Knapsack Problem with Conflicts, where certain items can not be packed together [see, e.g., 5, for a recent exact algorithm for the problem]. Nevertheless, this is a much wider-reaching constraint than the very local condition of not packing same-colored items adjacent. A recent work that is instead closer to ours is that of Malaguti et al. [10]. They study the Knapsack Problem with Group Fairness, where the items are partitioned into groups. Each item is associated with some resource consumption value and total resource consumption of the items in each group needs to be within some group-specific lower and upper bound. In this problem it is also possible to ensure that no group of items is packed more than all others as in our problem, but the number of items for each group needs to be decided apriori, while in our problem the balance must be kept for an arbitrary number of items per color. They propose a compact ILP model, as well as an extended set-partitioning ILP model. In the set-partitioning model, they select one packing of items for each group of items. They make the key observation that there is only a pseudo-polynomial number of relevant packings for each group, allowing them to completely enumerate the packings using a dynamic program and then solve the set-partitioning problem with an off-the-shelf ILP solver. One of the dynamic programs we propose in this article is based on a similar observation that for each color only a pseudo-polynomial number of packings is relevant; nevertheless, both the dynamic program we use to obtain these packings, as well as the overarching algorithm, is different from their approach due to the differing problem structure.

1.2. Paper outline and contributions

The remainder of this paper is organized as follows. In Section 2, we investigate the theoretical properties of the *Linear Programming* (LP) relaxation of ILP_{CKP}. We extend the classic results of the KP LP relaxation to this new variant, proving that there always exists an optimal solution to its LP relaxation with at most two fractional items. Furthermore, we analyze the relationship between the LP relaxations of the classical and colored problems, demonstrating that an optimal solution to the LP relaxation of ILP_{KP} provides information to identify a tight color constraint for an optimal solution to the LP relaxation of ILP_{CKP}. Leveraging these insights, we prove that the linear relaxation

of ILP_{CKP} can be solved in linear time, i.e., $\mathcal{O}(n)$, matching the theoretical complexity of the classical Knapsack Problem.

Section 3 introduces two exact *Dynamic Programming* (DP) algorithms for the CKP, establishing that the problem is weakly NP-hard. The first algorithm, detailed in Section 3.1, iterates item-by-item, resulting in a time complexity of $\mathcal{O}(b\,n^4)$. The second algorithm, presented in Section 3.4, adopts a decomposition-like approach that iterates color-by-color, solving an inner Knapsack Problem for each color and an outer recurrence to combine them, with a time complexity of $\mathcal{O}(b^2 n^3)$. We further enhance these approaches in Section 3.2 by deriving dominance rules to discard dominated states and fathoming rules that prune states which cannot be extended to optimal or feasible solutions.

Section 4 presents our computational experiments. We introduce a comprehensive benchmark of instances derived from the pricing subproblems of the CBPP, to cover a wide range of structural characteristics. We then evaluate the performance of both dynamic programming approaches, comparing them against the direct solution to the ILP_{CKP} formulation using state-of-the-art MIP solvers, in order to demonstrate their practical efficiency.

Finally, Section 5 summarizes our main contributions and outlines promising directions for future research.

2. Properties of the LP relaxation of ILP_{CKP}

The LP relaxation LP_{KP} of ILP_{KP} has some well-known properties, for which we derive analogous properties for the LP relaxation of ILP_{CKP}, denoted as LP_{CKP}:

$$(\text{LP}_{\text{CKP}}) \qquad \max \sum_{i \in \mathcal{I}} p_i x_i, \qquad\qquad (3a)$$

$$\sum_{i \in \mathcal{I}} w_i x_i \le b, \qquad\qquad (3b)$$

$$\sum_{i \in \mathcal{I}_c} x_i - \sum_{i \in \bar{\mathcal{I}}_c} x_i \le 1, \qquad \forall c \in \mathcal{C}, \qquad (3c)$$

$$x_i \le 1, \qquad \forall i \in \mathcal{I}, \qquad (3d)$$

$$x_i \ge 0, \qquad \forall i \in \mathcal{I}. \qquad (3e)$$

Specifically, we will characterize some optimal solutions to LP_{CKP}, show that if an optimal solution to LP_{KP} violates a color constraint, then there must be an optimal solution to LP_{CKP} where that color constraint is tight, and use both of these properties to prove that an optimal solution to LP_{CKP} can be found in linear time.

For this section, we extend the concepts of dominant and critical colors introduced in Section 1 to solutions of LP_{CKP}. Given a solution $\boldsymbol{x} \in [0,1]^n$ to LP_{CKP}, we say a color $c \in \mathcal{C}$ is dominant if $c \in \arg\max\limits_{c' \in \mathcal{C}} \sum\limits_{i \in \mathcal{I}_{c'}} x_i$, i.e., if it has the most (fractionally) packed items in solution. A dominant color c is also critical if $\sum\limits_{i \in \mathcal{I}_c} x_i - \sum\limits_{i \in \bar{\mathcal{I}}_c} x_i = 1$, i.e., if its associated color constraint is tight.

It is well known (see, e.g. [9]) that it is always possible to construct an optimal solution for LP_{KP} in which at most one variable takes on a fractional value. The corresponding item is called the *split item*. We show an analogous property for LP_{CKP}.

Lemma 1. *There is always an optimal solution $\boldsymbol{x}^*$ to LP_{CKP} in which at most two items are fractionally packed, i.e., $|\{i \mid x_i \in (0,1)\}| \leq 2$.*

Proof. Consider a feasible and bounded linear program in $\mathbb{R}^n$. Such a problem admits an optimal solution $\boldsymbol{x}^*$ that is an extreme point of the feasible solution polytope. Furthermore, at any extreme point in $\mathbb{R}^n$, at least n linearly independent constraints are active. In LP_{CKP}, constraints consist of one capacity constraint, a family of m color constraints, and bound constraints on the $\boldsymbol{x}$ variables. We observe that at most one color constraint can be active at any feasible solution to LP_{CKP}: hence, at most two non-bound constraints – namely, the capacity constraint and at most one color constraint – can be active at an extreme point. It follows that at least $n - 2$ of the remaining active constraints must be bound constraints. Since each bound constraint fixes one variable at either its lower or upper bound, and therefore to an integer value, at most two variables can be fractional at an extreme point solution. □

The two fractional items can be of any color, even the same non-critical color. We demonstrate this with the following example: consider a CKP instance with $n = 4$, $b = 10$, and $m = 2$; items 1 and 2 have color $c = 1$, $p_1 = p_2 = 100$, and $w_1 = w_2 = 4$; items 3 and 4 have instead color $c = 2$, $p_3 = 2, p_4 = 1$, and $w_3 = 4, w_4 = 1$. Evidently, in any optimal solution to LP_{CKP} we have that $x_1 = x_2 = 1$, but to fulfill the color constraint we need that $x_3 + x_4 \geq 1$, so that the color constraint associated with color 1 is not violated. Once x_1 and x_2 have been set to 1, with the remaining $b - w_1 - w_2 = 2$ capacity, the best feasible choice is to set $x_3 = \dfrac{1}{3}$ and $x_4 = \dfrac{2}{3}$, which fulfills both the color and capacity constraint at equality and packs as much of the third, more profitable item as possible.

Another interesting property of the regular KP is that an optimal solution to its LP relaxation can be found quickly by a combinatorial algorithm, specifically in $\mathcal{O}(n)$ time (see [9]). The same result can be

extended to LP$_{\text{CKP}}$. To show this, we need to know if there exists an optimal solution to LP$_{\text{CKP}}$ where a color is critical and, if yes, which it is. Note that, if no such color exists, we would obtain a feasible solution to LP$_{\text{CKP}}$ simply by solving the LP-relaxation of ILP$_{\text{KP}}$. As shown in Figure 1, if we solve an instance of the CKP without the color constraints and obtain a solution with a dominant color, it might still be that, considering the color constraints, all optimal CKP solutions have a different dominant color. In comparison, the LP-relaxation of the problem is much better behaved: indeed, finding an optimal solution to LP$_{\text{KP}}$, thus disregarding the color constraints, provides information about which color is critical in at least one optimal solution to LP$_{\text{CKP}}$, as detailed in the following Lemma.

Lemma 2. *Let $\boldsymbol{x}^*$ denote an optimal solution to the LP-relaxation of ILP$_{KP}$ where the color constraint is violated for some color $\tilde{c} \in \mathcal{C}$, i.e., $\sum_{i \in \mathcal{I}_{\tilde{c}}} x_i^* > \sum_{i \in \bar{\mathcal{I}}_{\tilde{c}}} x_i^* + 1$. Then, there always exists an optimal solution $\boldsymbol{x}'$ to LP$_{CKP}$ where $\tilde{c}$ is critical, i.e., $\sum_{i \in \mathcal{I}_{\tilde{c}}} x_i' = \sum_{i \in \bar{\mathcal{I}}_{\tilde{c}}} x_i' + 1$.*

Proof. Assume there exists an optimal solution $\boldsymbol{x}''$ to LP$_{\text{CKP}}$ where $\tilde{c}$ is not critical. Consider the convex combination $\boldsymbol{x}(\delta) = \delta\,\boldsymbol{x}^* + (1 - \delta)\,\boldsymbol{x}''$ for $\delta \in [0, 1]$. We observe that the color constraint associated to $\tilde{c}$ is satisfied at $\delta = 0$ and violated at $\delta = 1$, so by continuity there exists $\bar{\delta} \in (0, 1)$ where it holds at equality and such that $\boldsymbol{x}(\bar{\delta})$ is feasible for LP$_{\text{CKP}}$. Indeed, both $\boldsymbol{x}^*$ and $\boldsymbol{x}''$ satisfy all other constraints and so does any convex combination of them. Furthermore, since $\boldsymbol{x}''$ cannot have a greater value than $\boldsymbol{x}^*$ (i.e., $\boldsymbol{p}^\top \boldsymbol{x}'' \leq \boldsymbol{p}^\top \boldsymbol{x}^*$), it holds that

$$\boldsymbol{p}^\top \boldsymbol{x}(\bar{\delta}) = \boldsymbol{p}^\top \boldsymbol{x}'' + \delta\,(\boldsymbol{p}^\top \boldsymbol{x}^* - \boldsymbol{p}^\top \boldsymbol{x}'') \geq \boldsymbol{p}^\top \boldsymbol{x}''.$$

This proves that $\boldsymbol{x}(\bar{\delta})$ is also optimal for LP$_{\text{CKP}}$. $\qquad\square$

Lemma 2 establishes that a color $\tilde{c} \in \mathcal{C}$ that is critical in at least one optimal solution to LP$_{\text{CKP}}$ can be determined in $\mathcal{O}(n)$ time by solving LP$_{\text{KP}}$, if it exists. This allows to simplify the formulation of LP$_{\text{CKP}}$, discarding the color constraints associated with all colors $c \neq \tilde{c}$, while turning the color constraint associated to $\tilde{c}$ into an equality constraint. LP$_{\text{CKP}}$ thus reduces to the following linear program with two structural constraints:

$$\max \sum_{i \in \mathcal{I}} p_i\, x_i, \tag{4a}$$

$$\sum_{i \in \mathcal{I}} w_i\, x_i \leq b, \tag{4b}$$

$$\sum_{i \in \mathcal{I}} g_i\, x_i = 1, \tag{4c}$$

$$x_i \leq 1, \qquad \forall i \in \mathcal{I}, \qquad (4\text{d})$$

$$x_i \geq 0, \qquad \forall i \in \mathcal{I}, \qquad (4\text{e})$$

where g_i is a parameter equal to 1 if $\kappa_i = \tilde{c}$, and -1 otherwise.

We now state the main theoretical result of this section, showing that formulation (4) can be solved in linear time.

Proposition 1. *An optimal solution to the linear program* (4) *can be computed in* $\mathcal{O}(n)$ *time.*

Proof. We observe that (4) is a linear program with n variables subject to $2n$ simple bound constraints and exactly two structural constraints (Constraints (4b) and (4c)). The dual of this problem involves minimizing a convex piecewise linear function over a space of dimension $d = 2$, where the two dual variables correspond to the capacity and the color constraints, respectively. Since the dimension d is fixed and independent of the input size n, the problem falls into the class of low-dimensional linear programs solvable in linear time via the multidimensional search technique established by Megiddo [12]. Consequently, an optimal solution to (4) can be found in $\mathcal{O}(n)$ time. $\qquad\square$

A detailed description of the multidimensional search procedure of Megiddo [12] is beyond the scope of this paper; we refer the interested reader to the original article for further details, as well as to the works by Zemel [16] and Megiddo and Tamir [13], which discuss applications of this technique to problems sharing structural similarities with ours.

The results discussed in this section demonstrate that the introduction of color constraints does not increase the computational difficulty of the CKP LP-relaxation, compared to the standard KP. The crucial insight provided by Lemma 2 is that LP_{CKP} effectively collapses to a linear program with a fixed number of structural constraints (specifically, two) once a critical color (if it exists) is identified by solving the standard KP LP relaxation. This allows to extend the classic linear-time solvability property of LP_{KP} to the colored variant as well.

3. Two exact dynamic programming algorithms for solving the CKP

As discussed in Section 1, the CKP is at least weakly $\mathcal{NP}$-hard as a generalization of the KP. In this section, we provide two dynamic programming algorithms that run in pseudo-polynomial time, demonstrating that the problem is in fact also weakly $\mathcal{NP}$-hard. The first runs in $\mathcal{O}(b\,n^4)$ and the second in $\mathcal{O}(b^2\,n^3)$, so depending on the instance one or the other might have more promising performance.

Both algorithms make use of the following observation: to evaluate a solution, we do not actually need to know which colors are dominant, but just the number of items in a dominant color. So, in both approaches, a state does not keep track of the number of items in each color, but only considers the total number of items, the highest number of items in any color, and for the first algorithm also the number of items in the color of the current item. This information is always sufficient to determine if a state is feasible, and allows a smaller state space than explicitly tracking the number of items in each color as in one of the approaches proposed by Borges et al. [3].

To present our dynamic programming approaches more concisely, we introduce the following notation: For any item set $S \subseteq \mathcal{I}$, we denote by $p(S) = \sum_{i \in S} p_i$ the total profit of the set and by $w(S) = \sum_{i \in S} w_i$ its total weight. Moreover, recall that we denote the number of items of color $c \in \mathcal{C}$ in S by $k_c(S)$ (as introduced in Section 1). In the following, we will also denote by $\overline{k}_c(S) = \max_{c \in \mathcal{C}} k_c(S)$ the cardinality of a dominant color in S.

3.1. First dynamic program: iterating item by item

For our first approach, which we refer to as $\texttt{DP}_1$, we assume the items are sorted by color. We want to iteratively compute the function $f(i, t, d, a, q)$, that returns the optimal value for the CKP if we restrict the instance to the first i items, pack exactly t items in total, pack exactly d items in the dominant color, pack exactly a items of color κ_i (of the last considered item) and use exactly q capacity. More formally, we want to compute

$$f(i, t, d, a, q) = \max \left\{ p(S) \mid S \subseteq \{1, 2, \ldots, i\}, \ |S| = t, \ \overline{k}(S) = d, \ k_{\kappa_i}(S) = a, \ w(S) = q \right\}, \quad (5)$$

if such a solution exists. The optimal solution value for the entire instance can then be determined by finding $\max_{t, d \in \{0, 1, \ldots, n\}, \ 2d \leq t+1, \ a \in \{0, 1, \ldots, n\}, \ q \in \{0, 1, \ldots, b\}} f(n, t, d, a, q)$, i.e., finding the largest objective value among solutions that respect the color constraints.

For the sake of presentation, we describe the $\texttt{DP}_1$ algorithm as it would be implemented, instead of the classic recursive formula, which is hard to understand due to various necessary case distinctions. See Algorithm 1 for a pseudo-code description of $\texttt{DP}_1$.

Let $\sigma(a, d) : \mathbb{Z}_{\geq 0} \to \{0, 1\}$ be a function returning 1 if $d \leq a$, and 0 otherwise. We start with a state for the empty packing, i.e., $f(0, 0, 0, 0, 0) = 0$ and set $f(i, t, d, a, q) = -\infty$ otherwise. We then iterate through the items and extend all feasible states (Line 5) in two ways:

i) Packing item i (Line 12). In this case, the total number of items t increases by one, as does the number of items in the current color a. The number of items packed in the currently dominant

10

Algorithm 1 DP₁ algorithm for the CKP
___

1: **function** DP₁$(m, n, \boldsymbol{\kappa}, \boldsymbol{p}, \boldsymbol{w}, b)$

2:   $f(i, t, d, a, q) \leftarrow -\infty$ for all $i, t, d, a \in \{0, \ldots, n\}, q \in \{0, \ldots, w\}$ ▷ Initialize possible states

3:   $f(0, 0, 0, 0, 0) \leftarrow 0$ ▷ Initialize starting state

4:   **for** $i \in \{1, 2, \ldots, n\}$ **do**

5:     **for** t, d, a, q such that $f(i - 1, t, d, a, q) > -\infty$ **do**

6:       **if** $(i = 1) \vee (\kappa_i = \kappa_{i-1})$ **then**   ▷ Stay in same color

7:         $a' \leftarrow a$

8:       **else**   ▷ Change color – reset counter

9:         $a' \leftarrow 0$

10:       **end if**

11:       $s_{\text{pack}} = (i, \, t + 1, \, d + \sigma(a', d), \, a' + 1, \, q + w_i)$

12:       $f(s_{\text{pack}}) \leftarrow \max\{f(i - 1, t, d, a, q) + p_i, \, f(s_{\text{pack}})\}$ ▷ Pack item

13:       $f(i, t, d, a', q) \leftarrow \max\{f(i - 1, t, d, a, q), \, f(i, t, d, a', q)\}$ ▷ Do not pack item

14:     **end for**

15:   **end for**

16: **end function**
___

color increases only if the current color is dominant, i.e., if we have packed exactly $\overline{k}(S)$ items of the current color in the current solution S associated to the state, and thus $\sigma(a, d) = 1$. Finally, the weight of the packed items is increased by w_i. Then, we set the value of this new state to the maximum between its previously determined value and that of the state we extend plus the profit of the current item, namely p_i.

ii) Not packing item i (Line 13). Nothing changes about the packing except that we have considered one more item. The value of this next state is therefore set to the maximum between its previously determined value and that of the current state.

One special case occurs when we switch from one color to another, i.e., when $\kappa_i \neq \kappa_{i-1}$. In this case, we set the counter of items packed in the current color back to zero, namely $a' \leftarrow 0$ (Line 9).

The correctness of $\mathtt{DP_1}$ is formally stated in the following proposition.

Proposition 2. *The $\mathtt{DP_1}$ algorithm computes the value of $f(i, t, d, a, q)$ for all $i, t, d, a \in \{0, 1, \ldots, n\}$, $q \in \{0, 1, \ldots, b\}$ according to (5), if a corresponding solution exists, in at most $\mathcal{O}(b\,n^4)$ operations.*

Proof. For $i \in \{0, 1, \ldots, n\}$, $t, d, a \in \{0, 1, \ldots, n\}$, and $q \in \{0, 1, \ldots, b\}$, let $f^*(i, t, d, a, q)$ denote the maximum profit obtainable by selecting a subset of the first i items such that exactly t items are selected, the maximum number of selected items of any color equals d, exactly a selected items have color κ_i, and the total weight equals q. If no such subset exists, then we assume $f^*(i, t, d, a, q) = -\infty$. The $\mathtt{DP_1}$ algorithm maintains a table $f(i, t, d, a, q)$, and we prove that $f(i, t, d, a, q) = f^*(i, t, d, a, q)$ for all i, t, d, a, q by induction on i.

For $i = 0$, no items are available. The only feasible solution is the empty set, which satisfies $t = d = a = q = 0$, and has profit 0. Thus, $f^*(0, 0, 0, 0, 0) = 0$, and all other states are infeasible. The algorithm correctly initializes this state and assigns $-\infty$ to all others, so the claim holds for $i = 0$.

We now assume that, for some $i \geq 1$, the equality $f(i - 1, t, d, a, q) = f^*(i - 1, t, d, a, q)$ holds for all (t, d, a, q). We therefore want to show that $\mathtt{DP_1}$ correctly computes $f(i, \cdot)$. The algorithm iterates over all reachable states (t, d, a, q) at stage $i - 1$, i.e., those with $f(i - 1, t, d, a, q) > -\infty$, and determines the correct counter a' for the current color κ_i. If $i = 1$ or $\kappa_i = \kappa_{i-1}$, then the counter is preserved, so $a' = a$. Otherwise, since items are sorted by color, item i is the first occurrence of color κ_i, and the counter is correctly reset to $a' = 0$. From each such state, it considers the two exhaustive and mutually exclusive choices: excluding or including item i among the selected items. It follows that: i) if i is not selected, the selected set remains unchanged. The total number of selected items, the

number of items of each dominant color, and the total weight remain t, d, and q, respectively. Only the counter for the current color is updated to a'. The resulting state is therefore (i, t, d, a', q). The algorithm updates $f(i, t, d, a', q)$ by taking the maximum between its current value and the propagated value $f(i-1, t, d, a, q)$; ii) if item i is selected, the total number of selected items increases to $t+1$, the total weight to $q + w_i$, and the counter of the current color to $a' + 1$. The value of d must be updated accordingly. Since d is the maximum frequency among all colors in the partial solution at stage $i-1$, and the current color count was a', the new dominant frequency is $\max\{d, a'+1\}$. Because $d \geq a'$, this update can be written as $d + \sigma(a', d)$, where $\sigma(a', d) = 1$ if $a' = d$ and 0 otherwise. The algorithm thus transitions to the state $(i,\ t+1,\ d + \sigma(a', d),\ a'+1,\ q+w_i)$ and updates its value to $f(i-1, t, d, a, q) + p_i$, taking the maximum with any previously stored value.

To show that these transitions preserve optimality, consider an optimal solution to a given state (i, t, d, a, q), of value $f^*(i, t, d, a, q)$. Item i is either selected or not. If item i is not selected, the solution uses only items from $\{1, 2, \ldots, i-1\}$ and corresponds to a feasible state at stage $i-1$ with the same profit. By the inductive hypothesis, this value is stored in $f(i-1, \cdot)$ and is propagated to stage i by the "do not pack" transition. If item i is instead selected, removing it yields a feasible solution for the first $i-1$ items. This partial solution must be optimal for its parameters; otherwise, replacing it with a better one would improve the full solution, contradicting optimality. By the inductive hypothesis, its value is correctly stored in $f(i-1, \cdot)$, and the "pack" transition adds p_i and updates the state parameters consistently.

Since the algorithm considers both cases for every reachable state at stage $i-1$ and updates each destination state by taking the maximum profit over all incoming transitions, it follows that $f(i, t, d, a, q) = f^*(i, t, d, a, q)$ for all t, d, a, q. By induction, this holds for all $i \in \{0, 1, \ldots, n\}$.

The state space of DP$_1$ is defined by $i, t, d, a \in \{0, 1, \ldots, n\}$ and $q \in \{0, 1, \ldots, b\}$, and thus contains $\mathcal{O}(n^4 b)$ states. For each reachable state, the algorithm performs a constant number of operations. Consequently, the overall time complexity of DP$_1$ is $\mathcal{O}(n^4 b)$. □

We note that, to actually obtain an optimal CKP solution and not just the optimal solution value, we can store for each state the predecessor state that was extended to reach it, which then allows to backtrack and determine the packed items.

3.2. Improvements for DP$_1$

In this section, we propose some methods to improve the runtime of Algorithm 1 in practice.

First, note that many states permit no feasible solution (i.e., their value is never updated and remains equal to $-\infty$). Hence, instead of storing the full dynamic programming table for all values of (i, t, d, a, q), we propose to use a sparse data structure, like a hash map, and only store states that $\mathtt{DP_1}$ actually reaches. Second, we observe that some states can actually be merged or fathomed, because they do in fact represent identical partial solutions with respect to feasibility or can not be completed into optimal or feasible solutions.

3.2.1. Dominance rules

We observe that there may exist pairs of states such that one of these *dominates* the other, meaning that (given the same extension is applied to both the states) one will surely lead to a higher value than the other. It is therefore possible to add some dominance rule that stems from the fact that we so far computed the states as if we solved an exact knapsack problem, i.e., that the capacity constraint has to be fulfilled at equality.

DOMINANCE 1. Consider two states $s_1 = (i, t_1, d_1, a_1, q_1)$ and $s_2 = (i, t_2, t_2, a_2, q_2)$, such that

$$t_1 \geq t_2, \quad d_1 \leq d_2, \quad a_1 \leq a_2, \quad q_1 \leq q_2 \quad \text{and} \quad f(s_1) \geq f(s_2).$$

Then, s_1 dominates s_2, which can therefore be discarded. The reasoning is that every extension that can be applied to s_2 that results in a feasible solution can also be applied to s_1, since it is never further away from being feasible. Observe that s_1 has no less items packed in total and no more items packed in the dominant and current color, so it is not further away from fulfilling the color constraint than s_2. Additionally, s_1 has no less remaining capacity. Since it also gives at least the same profit, there is no need to keep state s_2.

For practical purposes, preliminary experiments showed that it is beneficial to only check this dominance criterion when a color was just finished and all values of a can be set to zero, so significantly less comparisons are necessary. For the same reason, we only check states with $t_1 = t_2$, instead of all those with $t_1 \geq t_2$.

DOMINANCE 2. The next group of state space reductions concerns situations in which two or more states are identical with regards to feasibility of the color constraints. First, consider the case that we switch to a new color. A state with $2d \leq t + 1$ is feasible with regards to the color constraint and all previously considered colors cannot become dominant anymore. So, at the first item of a new color, we can simply set the value of d in the new states to zero if $2d \leq t + 1$ for the state that is being extended. A variant of this reduction rule can also be applied within a color: if the number

of remaining items $\bar{n}$ in the current color is not sufficient to construct an infeasible solution, i.e., $2d \leq t + 1$ and $2(a + \bar{n}) \leq t + 1$, we can set $d = 0$ and $a = 0$. We can also set $a = 0$ if $a + \bar{n} \leq d$, because the current color cannot become dominant at all. By mapping d or a to zero, we merge multiple states and only keep the non-dominated ones.

3.2.2. Fathoming rules

In order to reduce the number of states that can be generated during the algorithm execution, it is also crucial to fathom some states based on optimistic bounds regarding the potential completions of the current partial solutions associated to them. Specifically, we fathom a state if it cannot become feasible or if it cannot achieve a better solution value than the best known primal bound with the remaining available items and capacity. To do so, we need to know the value

$$\bar{p}(i, q) = \max\{p(S) \mid S \subseteq \{i, i+1, \ldots, n\}, w(S) \leq q\},$$

which is the maximum profit achievable with the remaining items starting from item i and with q capacity or less, disregarding the color constraints. We also need to know the value

$$\bar{n}(i, q) = \max\{|S| \mid S \subseteq \{i, i+1, \ldots, n\}, w(S) \leq q\},$$

which is the maximum number of items that can be packed among the remaining ones starting from item i, and with q capacity or less, disregarding the color constraints. Note that both values can be precomputed in $\mathcal{O}(b\,n)$ operations for all (i, q) by solving a regular KP on the instance with the items ordered in reverse, using dynamic programming. We can therefore derive the following fathoming rules.

FATHOMING 1. Let LB denote any primal bound on the optimal CKP solution value. Then, if for a state $s = (i, t, d, a, q)$ it holds that if

$$f(s) + \bar{p}(i+1, b-q) \leq LB,$$

the state s cannot lead to any solution of value greater than LB and, therefore, it can be discarded.

FATHOMING 2. Let $i' = \min\{j \in \{i, i+1, \ldots, n\} \mid \kappa_j \neq \kappa_i\}$ denote the index of the first item whose color is different from κ_i. If for a state $s = (i, t, d, a, q)$ it holds that

$$2d > \begin{cases} t + 1 + \bar{n}(i+1, b-q), & \text{if } a < d, \\ t + 1 + \bar{n}(i', b-q), & \text{otherwise,} \end{cases}$$

15

then the state s cannot lead to any feasible solution and, therefore, it can be discarded.

We use the information we obtain from determining $\bar{\boldsymbol{p}}$ also in another way: for all $i \in \mathcal{I}$, $q \in \{0, 1, \ldots, b\}$, we determine the solution leading to the value of $\bar{p}(i, q)$; we then compute the total number of items $\bar{t}(i, q)$, the number of items in the dominant color $\bar{d}(i, q)$, and the number of items $\bar{a}(i, q)$ in color κ_i for this solution. Then, the following result holds.

FATHOMING 3. If for a state $s = (i, t, d, a, q)$ it holds that

$$\max \left\{ a + \bar{a}(i+1, b-q),\ d,\ \bar{d}(i+1, b-q) \right\} \leq t + \bar{t}(i+1,\ b-q) + 1,$$

then the primal bound LB is updated to $f(s) + \bar{p}(i+1, b-q)$, the combined solution is stored, and state s is fathomed.

3.3. Example for DP$_1$

Figure 2 shows the execution of DP$_1$, including the proposed improvements, on the instance introduced in Figure 1, with $n = 4$, $m = 2$, and $b = 10$. In the figure, each node represents a generated state identified by the tuple $s = (t, d, a, q)$, with the associated profit $f(s)$ displayed above the tuple. Solid arrows correspond to the decision of packing the current item, while dashed arrows represent skipping it. Fathomed states are highlighted in red, and report the reason why they have been fathomed below their associated node. The nodes are divided into layers, each corresponding to a different stage of the algorithm (and, therefore, to a different item considered for packing). The header of each layer shows the item index, along with its weight and profit in tuple format. Finally, the path highlighted in green indicates the sequence of decisions leading to the found optimal solution. Note that some of the states are discarded according to the first fathoming rule: as an example, take state $s = (1, 1, 1, 6)$ at Stage 3, which has a value of 15. This state cannot be extended to achieve a value greater than 18 (the current lower (primal) bound) and it is therefore fathomed by bound. For a visualization of the effect of the second fathoming rule, consider instead the state generated at Stage 2 by packing both items of color $c = 1$ (the bottom-most node). Although this partial solution yields a high profit, equal to 23, it is immediately fathomed. Indeed, since the capacity is fully utilized ($q = 10$), it is impossible to pack any subsequent items of color $c = 2$, required to restore feasibility. Consequently, the condition $2d \leq t + 1$ cannot be satisfied by any completion, and the state is discarded. Notice that, for the sake of presentation, we are not considering the effect of the third fathoming rule: indeed, if we applied such rule, the state $s = (1, 1, 1, 6)$, at Stage 2, would be immediately expanded to the optimal one, and discarded accordingly.
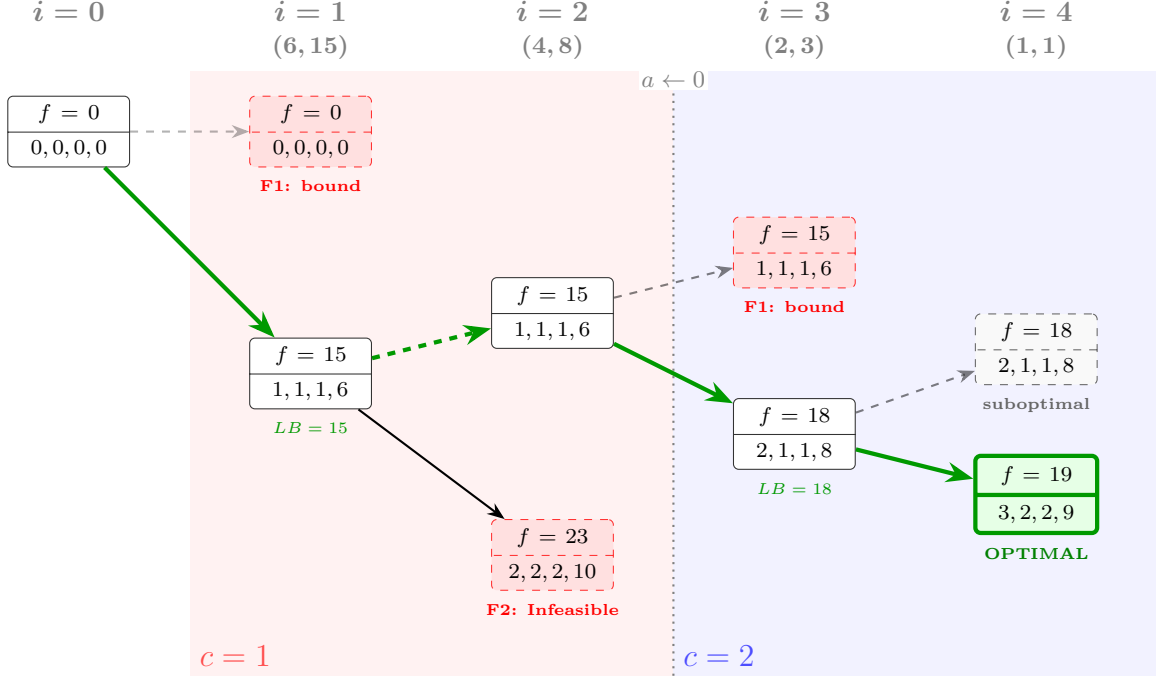
$i = 0$ $i = 1$ $(6, 15)$ $i = 2$ $(4, 8)$ $i = 3$ $(2, 3)$ $i = 4$ $(1, 1)$

$a \leftarrow 0$

$f = 0$ — $0, 0, 0, 0$

$f = 0$ — $0, 0, 0, 0$ — **F1: bound**

$f = 15$ — $1, 1, 1, 6$

$f = 15$ — $1, 1, 1, 6$

$f = 15$ — $1, 1, 1, 6$ — $LB = 15$

$f = 15$ — $1, 1, 1, 6$ — **F1: bound**

$f = 18$ — $2, 1, 1, 8$ — suboptimal

$f = 18$ — $2, 1, 1, 8$ — $LB = 18$

$f = 23$ — $2, 2, 2, 10$ — **F2: Infeasible**

$f = 19$ — $3, 2, 2, 9$ — **OPTIMAL**

$c = 1$ $c = 2$

Figure 2: Visual execution of DP_1 on the instance from Figure 1.

3.4. Second dynamic program: iterating color by color

The second dynamic programming algorithm, which we refer to as DP_2, is based on a decomposition idea. For each color $c \in \mathcal{C}$, we determine the best possible packing for every possible number of items of that color and capacity usage, and then we solve a variant of the Multiple-Choice Knapsack Problem [MCKP, see 9] where at most one packing can be selected for each color, while also adhering to the color constraints. To determine the relevant packings for a given color we utilize an 'inner' dynamic program, and then solve the modified MCKP with an 'outer' dynamic program.

The inner dynamic program determines the value

$$f_c(k, q) = \max\{p(S) \mid S \subseteq \mathcal{I}_c,\ w(S) = q,\ |S| = k\}, \tag{6}$$

which is the maximum profit obtainable if we solve a regular KP restricted to items in a given color $c \in \mathcal{C}$, using exactly $q \in \{0, 1, \dots, b\}$ capacity, and packing exactly $k \in \{0, 1, \dots, |\mathcal{I}_c|\}$ items. If no such packing exists for some (k, q), we set $f_c(k, q) = -\infty$. The values for all feasible $k \in \{0, 1, \dots, |\mathcal{I}_c|\}$ and $q \in \{0, 1, \dots, b\}$ can be determined by dynamic programming in at most $\mathcal{O}(b\,|\mathcal{I}_c|^2)$ operations per color (see, e.g., [9]).

For the outer dynamic program, we want to determine the optimal CKP solution value if we only consider the first c of the m colors, pack exactly t items, of which d are in each dominant color, and

use exactly q capacity. Let $\mathcal{S}_c = \{(k,q) \mid f_c(k,q) > -\infty\}$ denote the combinations of number of items k and capacity consumption q for which there exists a feasible packing in color $c \in \mathcal{C}$. Formally, we want to determine

$$f(c,t,d,q) = \max \left\{ \sum_{c'=1}^{c} f_{c'}(k_{c'}, q_{c'}) \mid \Big((k_1, q_1), (k_2, q_2), \ldots, (k_c, q_c)\Big) \in \mathcal{S}_1 \times \mathcal{S}_2 \times \cdots \times \mathcal{S}_c, \right. \tag{7}$$
$$\left. \sum_{c'=1}^{c} k_{c'} = t, \max_{c' \in \{1,2,\ldots,c\}} k_{c'} = d, \sum_{c'=1}^{c} q_{c'} = q \right\}$$

for all $c \in \{0, 1, \ldots, m\}$, $q \in \{0, 1, \ldots, b\}$, $t \in \{0, 1, \ldots, n\}$, and $d \in \{0, 1, \ldots, n\}$. We set $f(c, q, t, d)$ to $-\infty$ if no solution satisfies these conditions. Then, we obtain the value of an optimal CKP solution as $\max_{t, d \in \{1,2,\ldots,n\}, \, 2d \leq t+1, \, q \in \{1,2,\ldots,b\}} f(m, t, d, q)$.

As for DP_1, we explain this algorithm as it would be implemented instead of stating the corresponding recursion. See Algorithm 2 for a description in pseudo-code.

Algorithm 2 DP_2 algorithm for the CKP

1: **function** $\mathrm{DP}_2(m, n, \boldsymbol{\kappa}, \boldsymbol{p}, \boldsymbol{w}, b)$

2:     $f(c, t, d, q) \leftarrow -\infty$ for all $c \in \{0, 1, \ldots, m\}$, $t, d \in \{0, 1, \ldots, n\}$, $q \in \{0, 1, \ldots, b\}$

3:     $f(0, 0, 0, 0) \leftarrow 0$ ▷ Initialize starting state

4:     **for** $c \in \{1, 2, \ldots, m\}$ **do**

5:         determine $\mathcal{S}_c$ ▷ Solve inner dynamic program

6:         **for** t, d, q such that $f(c-1, t, d, q) > -\infty$ **do**

7:             **for** $(k, \Delta q) \in \mathcal{S}_c$ with $q + \Delta q \leq b$ **do**

8:                 $s_{\mathrm{new}} \leftarrow (c, \, t + k, \, \max\{d, k\}, \, q + \Delta q)$

9:                 $f(s_{\mathrm{new}}) \leftarrow \max\{f(s_{\mathrm{new}}), f(c-1, t, d, q) + f_c(k, \Delta q)\}$ ▷ Try packing

10:             **end for**

11:         **end for**

12:     **end for**

13: **end function**

For the initial state, we set $f(0, 0, 0, 0)$ to 0 and all other states to $-\infty$. Then we iterate through the colors and for the c-th color we extend each state $(c-1, t, d, q)$ with non-negative profit as follows: for each feasible packing in this color using Δq capacity and packing k items (i.e., $(k, \Delta q) \in \mathcal{S}_c$) that still fits into the knapsack (i.e, $\Delta q \leq b - q$), we determine the state we would reach if we selected this packing. Specifically, the total number of items increases by k; the number of items in the dominant color is the maximum of what it was before and k, i.e., d is only modified if we pack more items in

the current color than in any previous color; the capacity usage is increased by Δq. Note that we also consider the null-transition with $\Delta q = k = 0$, i.e., packing no items of the current color.

The correctness of $\mathtt{DP_1}$ is formally stated in the following proposition.

Proposition 3. *The* $\mathtt{DP_2}$ *algorithm computes the value of* $f(c, t, d, q)$ *for* $c \in \{0, 1, \ldots, m\}$, $t, d \in \{0, 1, \ldots, n\}$ *and* $q \in \{0, 1, \ldots, b\}$ *according to* (7), *if a corresponding solution exists, in at most* $\mathcal{O}(b^2 n^3)$ *operations.*

Proof. For $c \in \{0, 1, \ldots, m\}$, $t, d \in \{0, 1, \ldots, n\}$, and $q \in \{0, 1, \ldots, b\}$, let $f^*(c, t, d, q)$ denote the maximum profit obtainable by selecting items from the first c colors such that exactly t items are selected in total, the maximum number of selected items of any single color equals d, and the total weight equals q. If no such solution exists, we set $f^*(c, t, d, q) = -\infty$. The algorithm maintains a table $f(c, t, d, q)$. We prove that $f(c, t, d, q) = f^*(c, t, d, q)$ for all c, t, d, q by induction on c.

For $c = 0$, no colors are considered. The only feasible solution selects no items, has zero weight, and $d = 0$. Hence, $f^*(0, 0, 0, 0) = 0$, and all other states are infeasible. The algorithm initializes exactly this state and assigns $-\infty$ to all others, so the claim holds.

We now assume that $f(c-1, t, d, q) = f^*(c-1, t, d, q)$ holds for all states. Consider an optimal solution corresponding to $f^*(c, t', d', q')$. Such a solution selects a packing of color c consisting of k items with total weight Δq, where $(k, \Delta q) \in \mathcal{S}_c$. The remaining items form a feasible solution over the first $c - 1$ colors with parameters $t = t' - k$, $q = q' - \Delta q$, and $d = \max_{c' < c} k_{c'}(S)$. The value of d of the full solution is therefore $d' = \max\{d, k\}$.

By optimality, the partial solution over the first $c - 1$ colors must be optimal for the parameters $(c-1, t, d, q)$; otherwise, replacing it with a better one would increase the total profit. By the induction hypothesis, its profit equals $f(c - 1, t, d, q)$. Moreover, the packing chosen for color c must be optimal among all packings of k items with weight Δq; otherwise, it could be replaced by a strictly better packing, contradicting the correctness of the inner dynamic program.

Thus, the total profit of the solution equals $f(c - 1, t, d, q) + f_c(k, \Delta q)$, and the algorithm considers this transition and updates the state $(c, t+k, \max\{d, k\}, q+\Delta q)$. Since the algorithm maximizes over all feasible predecessors and all feasible packings in $\mathcal{S}_c$, it follows that $f(c, t', d', q') = f^*(c, t', d', q')$. This completes the inductive proof.

The total running time of $\mathtt{DP_2}$ is the sum of the costs of the inner and outer dynamic programs. For a fixed color c, the inner dynamic program computes the table $f_c(k, q)$ in $\mathcal{O}(|\mathcal{I}_c|^2 b)$ time. Summing over

all colors and observing that $\sum_{c=1}^{m} |\mathcal{I}_c|^2 \leq \left(\sum_{c=1}^{m} |\mathcal{I}_c| \right)^2 = n^2$, the total time for the inner phase is $\mathcal{O}(n^2 b)$. The outer dynamic program consists instead of m stages. At each stage c, the state space (t, d, q) has size $\mathcal{O}(n^2 b)$. For each state, the algorithm iterates over all feasible packings in $\mathcal{S}_c$, whose size is bounded by $\mathcal{O}(|\mathcal{I}_c| b)$. Thus, the time required at stage c is $\mathcal{O}(n^2 b \cdot |\mathcal{I}_c| b) = \mathcal{O}(n^2 b^2 |\mathcal{I}_c|)$. Summing over all colors yields a total time complexity of

$$\mathcal{O}\left(n^2 b^2 \sum_{c=1}^{m} |\mathcal{I}_c| \right) = \mathcal{O}(n^3 b^2).$$

$\square$

3.5. Improvements for the second dynamic program

Most of the improvements from $\mathtt{DP_1}$ can also be applied to $\mathtt{DP_2}$. Specifically, we also use a sparse data structure and do the following reductions:

i) For two states $s_1 = (i, t, d_1, q_1)$ and $s_2 = (i, t, d_2, q_2)$ with $q_1 \leq q_2$, $d_1 \leq d_2$ and $f(s_1) \geq f(s_2)$, it is possible to discard s_2.

ii) For a state (i, t, d, q) with $2d \leq t + 1$, we can set d to 0 because the dominant color among those considered so far is not critical and cannot become so anymore.

We also fathom states that cannot become feasible or optimal with the remaining items. Specifically, let $\bar{p}(i, q)$ and $\bar{n}(i, q)$ be the best possible profit and highest possible number of items that can be packed if we only consider colors from the i-th and using at most q units of capacity. Then, we fathom a state $s = (i, t, d, q)$ if $f(s) + \bar{p}(i + 1, b - q) < LB$ for some primal bound LB or if $2d > t + 1 + \bar{n}(i + 1, b - q)$. We use the same criteria to discard states in the inner DP that can never be part of a feasible or optimal solution when considering the items of all other colors.

3.6. Example for $\mathtt{DP_2}$

Figure 2 shows the execution of $\mathtt{DP_2}$ on the instance introduced in Figure 1, with $n = 4$, $m = 2$, and $b = 10$. In the figure, each node represents a generated state identified by the tuple $s = (t, d, q)$, with the associated profit $f(s)$ displayed above the tuple. The nodes are divided into layers, each corresponding to a different stage of the algorithm (and, therefore, to a different color). Solid arrows correspond to the decision of packing at least one item of the current color, while dashed arrows represent the choice of packing none of its items. Above each arc we show the set of items packed in the next color and the corresponding profit. In the upper section of each layer, the figure shows the packing options returned by the 'inner' dynamic program restricted to the sole items of that color.

For each option, we list the corresponding set of items S, number of items $k = |S|$, capacity usage Δq and profit $f_c(k, \Delta q)$. All the remaining elements of the figure have the same meaning as in Figure 2.
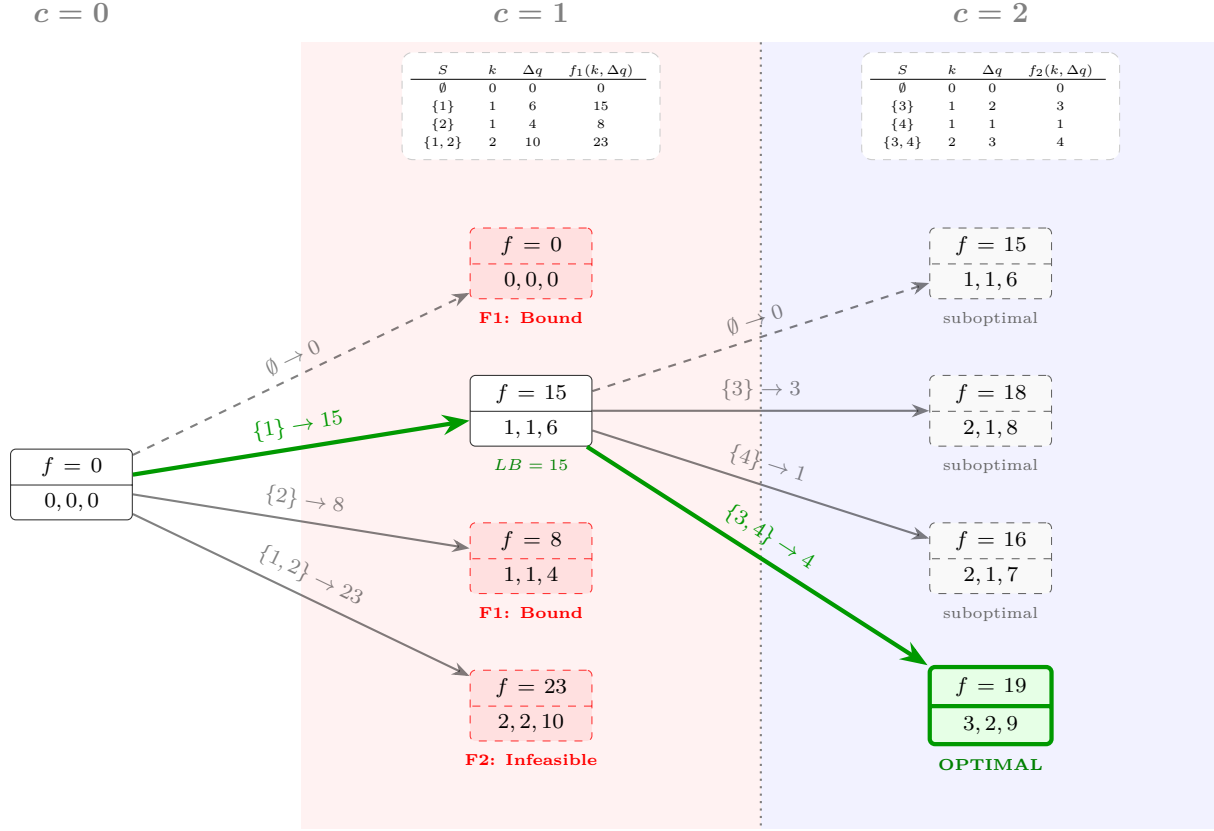

Figure 3: Visual execution of DP$_2$ on the instance from Figure 1.

4. Computational experiments

In this section, we present the results of our computational experiments, aimed at evaluating the performance of the two exact Dynamic Programming algorithms for the CKP, DP$_1$ and DP$_2$, introduced in Sections 3.1 and 3.4, respectively. Our experiments pursue two main goals: first, to assess the computational effectiveness of the proposed dynamic programming approaches by comparing them against the direct solution of the ILP$_{\text{CKP}}$ formulation using state-of-the-art MIP solvers; second, to demonstrate the practical efficiency of the algorithms on instances covering a wide range of structural characteristics.

To the best of our knowledge, the CKP is a novel generalization of the Knapsack Problem that has not been previously studied in the literature (see Section 1.1). Consequently, no publicly available benchmark instances exist. For this reason, we introduce a comprehensive benchmark of instances

derived from the pricing subproblems of the Colored Bin Packing Problem (CBPP), which we describe in Section 4.1.

4.1. Library of benchmark CKP instances

In this section, we introduce the library of benchmark instances specifically designed for the CKP. This testbed aims to provide a diverse and representative collection of instances, varying in size and structural characteristics, with the goal of comparing on such dataset the proposed methods and obtaining insights about how different instance features influence computational performance.

As discussed in Section 1.1, the CKP arises as pricing problem of the CBPP (see [3]). Thus, we decided to generate CKP instances by solving the root node of CBPP instances by column generation. As a result, we obtained a large set of CKP instances for each CBPP instance. All CKP instances corresponding to the same CBPP instance share the same number of items, the items sizes, the item colors, and the capacity of the knapsack. In contrast, the profits of the items differ as they stem from the dual solutions of the master problem and are used to compute the reduced costs of columns. Since $\boldsymbol{p} \in \mathbb{Z}^n$, we scaled the dual variable values and set $p_i = \lfloor 10^5 \pi_i \rfloor$ for $i \in \mathcal{I}$ and π_i being the corresponding dual variable value. For our computational campaign, we selected a subset of the instances proposed by Borges et al. [3]. Specifically, we used the uniform randomly generated instances, the randomly generated instances with zipf distribution of colors, and the ANI instances with 201 items and 2 colors, which were taken from the BPP literature. For the specific used parameters we refer to Borges et al. [3].

Since our approaches use bounds obtained by solving a Knapsack Problem for each instance, we check if the returned KP solution is feasible for the CKP as well. Instances for which the optimal solution to the standard Knapsack Problem was already feasible for the CKP (i.e., it satisfied the color constraints) were removed from the original set because they can be easily solved using any state-of-the-art knapsack solver. The results we present in the following section are therefore just based on instances where our implementation did not find a knapsack solution that is feasible for the CKP.

4.2. Computational performance

In this section, we assess the computational performance of our proposed dynamic programming algorithms, `DP`$_1$ and `DP`$_2$, by comparing them against the direct solution of the ILP$_{\text{CKP}}$ formulation using two state-of-the-art MIP solvers: the commercial solver `IBM ILOG CPLEX` [8] and the free and open-source solver `SCIP` [7]. All tests were conducted on Debian 11 computing nodes equipped with

two Intel Xeon L5630 processors with at least 16 GB of DDR3 RAM. We used the `C++` interfaces of `IBM ILOG CPLEX v22.1.0` and `SCIP v10.0.0`, running in single-thread mode with default settings. Moreover, we implemented our algorithms in `C++`.

Table 1 summarizes the results on the set of instances of the uniform family. The table reports the performance metrics on a large set of uniformly randomly generated instances. The instances are categorized by the number of items n, the capacity b, the number of color classes m, and the weight generation interval W ($W_1 = [0.1, 0.8]$ and $W_2 = [0.01, 0.25]$ relative to capacity). The column "# inst." reports the number of instances in the test set without the trivial (i.e., KP-solveable) instances. The column "% rem." indicates the percentage of of overall instances that were removed for being trivial. Finally, the last four columns report the average solving times in milliseconds for the four approaches on these remaining instances. Bold values indicate the lowest average computational time.

The data in Table 1 highlights several key insights: first, the "% rem." column reveals a strong inverse correlation between the number of color classes (m) and the restrictiveness of the color constraints. When m is high (e.g., $m = 15$), the probability of selecting consecutive items of the same color is lower. Consequently, for instances with $m = 15$, up to 99.9% of the generated instances were removed because the standard knapsack solution was already valid. Conversely, for $m = 2$, the color constraint is highly restrictive, with removal rates dropping to approximately 20–35%, which suggests that, for the majority of these instances, the color constraints are binding.

Regarding computational performance, the proposed DP algorithms significantly outperform the general-purpose solvers on the set of non-trivial uniform instances. DP_1 is the fastest approach in 29 out of 30 classes, with an overall average time of 14 ms. This is orders of magnitude faster than `CPLEX` (1425 ms) and `SCIP` (4294 ms). The difficulty of the instances for the MIP solvers appears to be driven by the weight structure (W_1 vs. W_2). Instances with W_1 (weights in $[0.1, 0.8]$) are harder for branch-and-bound solvers, with `CPLEX` and `SCIP` taking on average multiple seconds for instances with $n = 1,000$. In contrast, our DP algorithms remain extremely efficient on these inputs, often solving them in under 5 ms. The only case where `CPLEX` is competitive with DP_1 is for $n = 1,000$ with $m = 2$ and W_2 weights, where it achieves a marginally better time (62.7 ms vs 63.8 ms). However, DP_1 remains efficient across the entire range of parameters, which proves it is the most efficient approach for solving instances of this family.

Table 2 summarizes instead the results on the benchmark set of instances with zipf distribution of colors. Similarly to Table 1, this table reports the instance parameters (n, b), the number of instances

Table 1: Results for the uniform randomly generated instances. For each class of instances we give the number of items n, the capacity b, the number of colors m, and the interval W from which the item size (as a proportion of b) is sampled ($W_1 = [0.1, 0.8], W_2 = [0.01, 0.25]$). We further state the number of instances after removal, what percentage of them were removed because the standard knapsack solution was feasible for the CKP, and the average running time in milliseconds for each approach.

Instance class						Solving time (ms)			
n	b	m	W	# inst.	% rem.	DP_1	DP_2	CPLEX	SCIP
300	500	2	W_1	5727	35.1	**1.5**	3.0	186.6	1130.4
			W_2	6325	20.5	**21.2**	64.9	34.0	292.7
		7	W_1	2289	79.1	**1.2**	4.5	1132.0	5118.7
			W_2	112	98.6	**1.3**	16.0	37.7	262.2
		15	W_1	1458	86.0	**1.2**	2.9	1545.3	7497.6
			W_2	14	99.8	**1.2**	26.0	27.4	187.1
	750	2	W_1	6533	32.3	**2.1**	5.1	296.7	1503.1
			W_2	6641	19.8	**19.9**	83.0	42.5	358.9
		7	W_1	2175	79.3	**1.8**	5.9	1254.5	5494.6
			W_2	102	98.8	**2.0**	35.8	39.7	331.3
		15	W_1	1404	86.6	**1.8**	4.2	1493.3	6802.6
			W_2	8	99.9	**1.9**	19.9	25.9	135.6
500	500	2	W_1	12 014	33.0	**2.4**	5.5	442.0	2400.9
			W_2	10 815	22.2	**21.2**	62.8	43.3	413.0
		7	W_1	4108	78.8	**2.1**	7.7	6414.8	10 906.5
			W_2	176	98.8	**2.2**	25.8	48.7	426.2
		15	W_1	2425	86.6	**2.1**	6.8	9021.6	19 130.9
			W_2	19	99.9	**2.1**	38.5	50.7	359.2
	750	2	W_1	12 830	33.2	**4.1**	8.8	886.2	2934.2
			W_2	11 140	23.7	**22.8**	95.7	53.3	493.0
		7	W_1	4347	78.1	**3.5**	12.4	3756.6	13 897.3
			W_2	185	98.7	**3.6**	46.5	59.5	547.7
		15	W_1	2593	87.3	**3.4**	10.5	6381.5	18 692.0
			W_2	18	99.9	**3.5**	58.7	64.5	593.1
	1000	2	W_1	13 104	32.0	**5.4**	11.9	1113.6	3111.3
			W_2	12 095	21.8	63.8	214.8	**62.7**	560.2
		7	W_1	5078	74.4	**4.9**	17.8	5536.2	15 336.5
			W_2	162	99.0	**4.9**	69.0	65.3	567.3
		15	W_1	3634	81.9	**4.8**	14.5	7245.5	22 216.5
			W_2	22	99.9	**4.8**	109.0	49.9	422.8
Tot/Avg				127 553	69.4	**14.0**	47.3	1425.8	4294.1

Table 2: Results for the randomly generated instances with zipf distribution of colors. Given are the parameters of each class of instances, how many instances they comprise (after removal), and how many instances were removed because the standard knapsack solution was feasible for the CKP. In addition, the average running times in milliseconds of each approach for each class of instances are shown. n corresponds to the number of items, b to the capacity.

Instance class				Solving time (s)			
n	b	# inst.	% rem.	DP_1	DP_2	CPLEX	SCIP
300	300	7012	7.0	173.7	128.2	**42.0**	313.7
	500	7630	4.4	331.5	308.1	**53.0**	400.0
	750	8399	3.4	719.5	885.4	**64.3**	472.4
500	300	12 466	2.8	535.3	404.9	**62.9**	505.2
	500	13 227	3.5	1017.2	973.8	**80.3**	639.4
	750	14 585	2.1	2102.4	2523.8	**97.9**	785.4
Tot/Avg		63 319	3.6	956.8	1033.2	**71.3**	559.6

before filtering ("# inst."), the percentage of removed instances ("% rem."), and the average solving times for the four competing algorithms. The results in Table 2 present a different landscape compared to the uniform distribution case. Here, CPLEX consistently outperforms all other methods, achieving the lowest average solving times across all instance classes, with an overall average slightly above 71 ms. In contrast, DP_1 and DP_2 require significantly more time, with an average solving time of roughly 1 second per instance. This performance shift can be attributed to the structural properties induced by the color distribution. When items are concentrated in a few dominant color classes, the effective state space for the DP algorithms may remain large, as there are many items of the same color that can potentially be packed. On the other hand, the color constraints may become more binding in a branch-and-bound context. Since a large portion of items share the same color, the branching decisions quickly lead to infeasibility when attempting to pack many of them, allowing the solvers to prune the branching tree more aggressively.

Furthermore, the "% rem." column indicates that very few instances (only 2–7%) were removed. This confirms that the standard knapsack solution is rarely feasible for these instances, implying that there is almost always an active color constraint. Nevertheless, it is worth noting that while slower than CPLEX, the DP algorithms still maintain reasonable solving times (typically under 2 seconds), remaining a viable option even for this class of instances. Furthermore, its performance is still comparable to that of SCIP, especially for smaller instances with $b \leq 500$: specifically, for the group of smallest instances ($n = 300$ and $b = 300$), DP_1 and DP_2 have an average solving time which is around half of that of SCIP.

To conclude our experimental analysis on different families of instances, we also evaluated the algorithms on the ANI class, a dataset comprising $50\,513$ instances after $29.6\,\%$ were removed. The results confirm the trends observed for the uniformly generated instances. $\mathtt{DP_1}$ confirms to be the most performing algorithm, solving these instances with an average time of just $11\,\mathrm{ms}$. This performance is approximately 36 times faster than $\mathtt{CPLEX}$, which requires an average time slightly above $400\,\mathrm{ms}$, and orders of magnitude faster than $\mathtt{SCIP}$ ($1653\,\mathrm{ms}$ on average). $\mathtt{DP_2}$ remains competitive with an average time of $80\,\mathrm{ms}$, though less efficient than $\mathtt{DP_1}$.

Finally, Figure 4 illustrates the average solving time required by the four approaches ($\mathtt{DP_1}$, $\mathtt{DP_2}$, $\mathtt{CPLEX}$, and $\mathtt{SCIP}$) to solve the pricing subproblems generated during the solution of the CBPP at the root node of the branch-and-price tree. The x-axis represents the progress of the column generation procedure, expressed as the percentage of pricing problems solved, while the y-axis reports the average computational time taken by the different solution algorithms.

A clear increasing trend in difficulty is observable for all methods as the column generation progresses. This behavior is typical in branch-and-price algorithms: in the early iterations, the dual values often distribute unevenly, making it easier for solvers to find optimal columns or prune the search space. As the column generation converges, the dual variables stabilize, and the distinction between promising and non-promising items becomes less pronounced, typically requiring a more exhaustive search to certify optimality. However, the impact of this increasing difficulty varies significantly between the approaches. The MIP solvers, particularly $\mathtt{SCIP}$, exhibit a more significant growth in solving times, which shows a greater sensitivity to the specific distribution of item profits. In contrast, the DP algorithms, particularly $\mathtt{DP_1}$, demonstrate a greater stability. This is consistent with the design of $\mathtt{DP_1}$: its computational complexity is indeed primarily determined by capacity and number of items, parameters whose value remains constant throughout the column generation process. Albeit the fathoming rules may become less effective as the item profit distribution varies, this does not significantly affect the performance of $\mathtt{DP_1}$, whose profile remains below that of all other solution approaches for almost the entire x-axis.

5. Conclusions

In this paper, we introduced and analyzed the Colored Knapsack Problem (CKP), a novel generalization of the classical Knapsack Problem where items are partitioned into color classes and must be packed avoiding consecutive items of the same color. From a theoretical standpoint, we established the weak NP-hardness of the problem and analyzed its linear programming relaxation. We proved that an
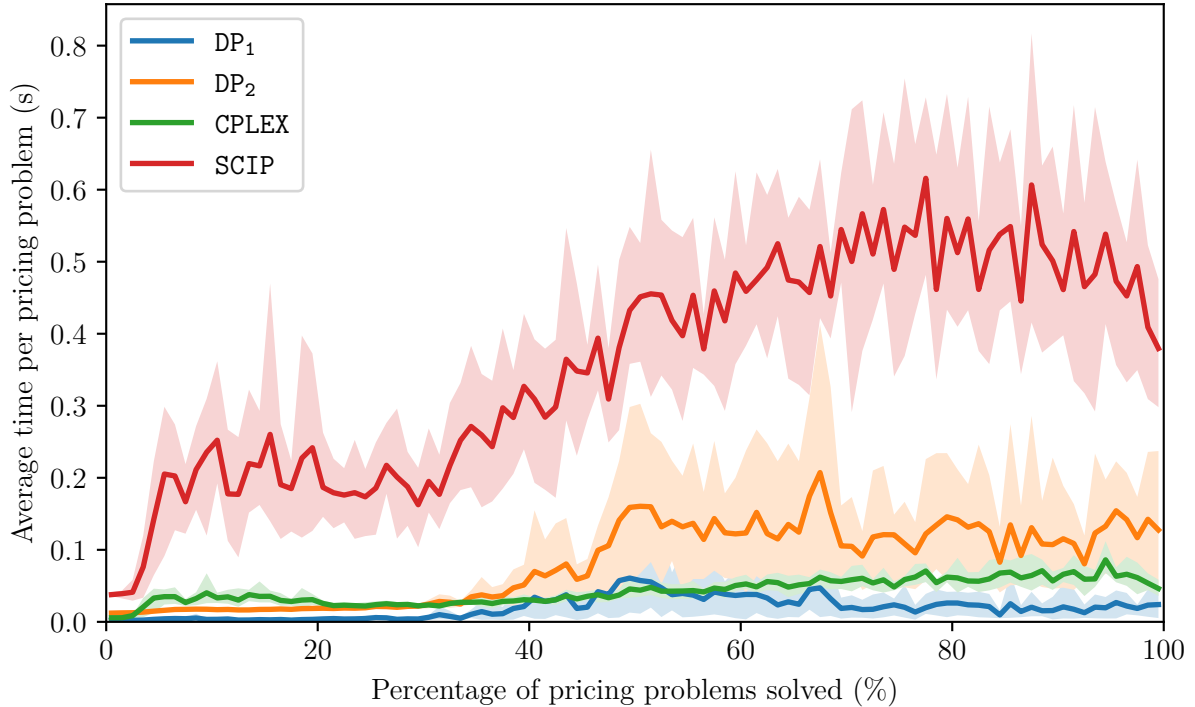
Figure 4: Evolution of the average solving time (in seconds) required by the four approaches to solve the pricing subproblems generated during the solution of the CBPP at the root node of the branch-and-price tree.

optimal solution to the relaxation always exists with at most two fractional items and demonstrated that it can be found in $\mathcal{O}(n)$ time, matching the complexity of the classical KP.

From the computational perspective, we instead proposed two pseudo-polynomial time dynamic programming algorithms, $\mathtt{DP_1}$ and $\mathtt{DP_2}$, with time complexities of $\mathcal{O}(bn^4)$ and $\mathcal{O}(b^2n^3)$, respectively. We further enhanced these algorithms with specific dominance and fathoming rules to reduce the search space. Our computational experiments, performed on a newly introduced benchmark set of instance derived from the Colored Bin Packing Problem, showed that our dynamic programming approaches significantly outperform state-of-the-art general-purpose MIP solvers on most of the instance families.

Future research avenues naturally include the development of approximation algorithms or fully polynomial-time approximation schemes (FPTAS) for the CKP, as well as the design of combinatorial branch-and-bound algorithms which leverage the low computational effort required to find an optimal solution to the CKP linear relaxation. A polyhedral study of the problem also appears promising, specifically to derive new valid inequalities or to extend classic cuts, such as cover cuts, to this knapsack variant. Furthermore, given the close relationship with the Colored Bin Packing Problem, incorporating the proposed algorithms as pricing solvers within a branch-and-price framework for the

CBPP constitutes a promising direction for future study.

Acknowledgments

Statement on AI Usage in This Work

We attempted to utilize commercially available LLMs like Gemini and ChatGPT to generate ideas for Dynamic Programming approaches to this problem but the proposed algorithms were either asymptotically slower than our ideas (e.g., using one resource per color to track the number of items packed in that color) or not correct. For solving the LP relaxation of the CKP we originally developed a cubic time algorithm. Once we realized that the results of [12] in combination with the property that we can "predict" the dominant color in the LP may lead to a linear time algorithm, the same LLMs were used to find appropriate sources and develop the proof idea. All text in this manuscript was written by us without any AI assistance and all citations were checked manually by us. We used GitHub Copilot for support in implementing our proposed algorithms and integer programming models as well as creating plots and tables. All AI-generated code was manually checked for accuracy by us.

References

[1] J. Balogh, J. Békési, G. Dosa, H. Kellerer, and Z. Tuza. Black and white bin packing. In T. Erlebach and G. Persiano, editors, *Approximation and Online Algorithms*, pages 131–144, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38016-7.

[2] J. Balogh, J. Békési, G. Dósa, L. Epstein, H. Kellerer, A. Levin, and Z. Tuza. Offline black and white bin packing. *Theoretical Computer Science*, 596:92–101, Sept. 2015. ISSN 0304-3975. doi: 10.1016/j.tcs.2015.06.045. URL https://www.sciencedirect.com/science/article/pii/S0304397515005551.

[3] Y. G. F. Borges, R. C. S. Schouery, and F. K. Miyazawa. Mathematical models and exact algorithms for the colored bin packing problem. *Comput. Oper. Res.*, 2024.

[4] V. Cacchiani, M. Iori, A. Locatelli, and S. Martello. Knapsack problems — an overview of recent advances. part i: Single knapsack problems. *Computers & Operations Research*, 143:

105692, 2022. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2021.105692. URL `https://www.sciencedirect.com/science/article/pii/S0305054821003877`.

[5] S. Coniglio, F. Furini, and P. S. Segundo. A new combinatorial branch-and-bound algorithm for the knapsack problem with conflicts. *Eur. J. Oper. Res.*, 289(2):435–455, 2021.

[6] G. Dósa and L. Epstein. Colorful bin packing. In R. Ravi and I. L. Gørtz, editors, *Algorithm Theory – SWAT 2014*, pages 170–181, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08404-6.

[7] C. Hojny, M. Besançon, K. Bestuzheva, S. Borst, A. Chmiela, J. Dionísio, L. Eifler, M. Ghannam, A. Gleixner, A. Göß, A. Hoen, R. van der Hulst, D. Kamp, T. Koch, K. Kofler, J. Lentz, S. J. Maher, G. Mexi, E. Mühmer, M. E. Pfetsch, S. Pokutta, F. Serrano, Y. Shinano, M. Turner, S. Vigerske, M. Walter, D. Weninger, and L. Xu. The SCIP Optimization Suite 10.0. Technical report, Optimization Online, November 2025. URL `https://optimization-online.org/2025/11/the-scip-optimization-suite-10-0/`.

[8] IBM Corporation. *IBM ILOG CPLEX Optimization Studio*. IBM, 2023. URL `https://www.ibm.com/products/ilog-cplex-optimization-studio`. Version 22.1.1.

[9] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin Heidelberg, 2004.

[10] E. Malaguti, P. Paronuzzi, and A. Santini. Algorithms and complexity results for the 0–1 knapsack problem with group fairness. *Optimization Letters*, 2025.

[11] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, New York, 1990.

[12] N. Megiddo. Linear programming in linear time when the dimension is fixed. *Journal of the ACM (JACM)*, 31(1):114–127, 1984.

[13] N. Megiddo and A. Tamir. Linear time algorithms for some separable quadratic programming problems. *Operations Research Letters*, 13(4):203–211, 1993.

[14] A. A. Pessoa, R. Sadykov, E. Uchoa, and F. Vanderbeck. A generic exact solver for vehicle routing and related problems. *Math. Program.*, 183(1):483–523, 2020.

[15] J. Valério De Carvalho. Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 86:629 – 659, 1999.

doi: 10.1023/a:1018952112615. URL `https://www.scopus.com/inward/record.uri?eid=2-s2.0-0042094654&doi=10.1023%2fa%3a1018952112615&partnerID=40&md5=a192eb4fbe711a1c49d8da1b449e03fd`. Cited by: 211.

[16] E. Zemel. An o(n) algorithm for the linear multiple choice knapsack problem and related problems. *Information Processing Letters*, 18(3):123–128, 1984.