A Fast Combinatorial Algorithm for the Bilevel Knapsack Problem with Interdiction Constraints

Noah Weninger(✉) and Ricardo Fukasawa

University of Waterloo, Waterloo, ON, Canada
{nweninger,rfukasawa}@uwaterloo.ca

Abstract. We consider the bilevel knapsack problem with interdiction constraints, a fundamental bilevel integer programming problem which generalizes the 0-1 knapsack problem. In this problem, there are two knapsacks and n items. The objective is to select some items to pack into the first knapsack such that the maximum profit attainable from packing some of the remaining items into the second knapsack is minimized. We present a combinatorial branch-and-bound algorithm which outperforms the current state-of-the-art solution method in computational experiments by 4.5 times on average for all instances reported in the literature. On many of the harder instances, our algorithm is hundreds of times faster, and we solved 53 of the 72 previously unsolved instances. Our result relies fundamentally on a new dynamic programming algorithm which computes very strong lower bounds. This dynamic program solves a relaxation of the problem from bilevel to $2n$-level where the items are processed in an online fashion. The relaxation is easier to solve but approximates the original problem surprisingly well in practice. We believe that this same technique may be useful for other interdiction problems.

Keywords: Bilevel programming · Interdiction · Knapsack problem · Combinatorial algorithm · Dynamic programming · Branch and bound

1 Introduction

Bilevel integer programming (BIP), a generalization of integer programming (IP) to two-round two-player games, has been increasingly studied due to its wide real-world applicability [5,12,17]. In the BIP model, there are two IPs, called the *upper level* (or *leader*) and *lower level* (or *follower*), which share some variables between them. The objective is to optimize the upper level IP but with the constraint that the shared variables must be optimal for the lower level IP. The term *interdiction* is used to describe bilevel problems in which the upper level IP has the capability to block access to some resources used by the lower level IP. The upper level is typically interested in blocking resources in a way that produces the worst possible outcome for the lower level IP. For instance, the resources may be nodes or edges in a graph, or items to be packed into a knapsack. These problems often arise in military defense settings (e.g., see [17]).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
A. Del Pia and V. Kaibel (Eds.): IPCO 2023, LNCS 13904, pp. 438–452, 2023.
https://doi.org/10.1007/978-3-031-32726-1_31

In this paper we study the bilevel knapsack problem with interdiction constraints (BKP), which was introduced by DeNegre in 2011 [6]. This problem is a natural extension of the 0-1 knapsack problem (KP) to the bilevel setting. Formally, we are given n items. Each item $i \in \{1,\ldots,n\}$ has an associated profit $p_i \in \mathbb{Z}_{>0}$, upper-level weight $w_i^U \in \mathbb{Z}_{>0}$ and lower-level weight $w_i^L \in \mathbb{Z}_{\geq 0}$. The upper-level knapsack has capacity $C^U \in \mathbb{Z}_{\geq 0}$ and the lower-level knapsack has capacity $C^L \in \mathbb{Z}_{\geq 0}$. We use the standard notation: for a vector x and set S we let $x(S) := \sum_{i\in S} x_i$. The problem BKP can then be stated as follows:

$$\min_{X\in\mathcal{U}} \max_{Y\in\mathcal{L}(X)} p(Y) \qquad \text{(objective)}$$

$$\text{where } \mathcal{U} = \left\{X \subseteq \{1,\ldots,n\} : w^U(X) \leq C^U\right\}, \qquad \text{(upper level)}$$

$$\text{and } \mathcal{L}(X) = \left\{Y \subseteq \{1,\ldots,n\} \setminus X : w^L(Y) \leq C^L\right\}. \qquad \text{(lower level)}$$

We call a solution (X,Y) *feasible* if $X \in \mathcal{U}$ and $Y \in \text{argmax}\{p(\hat{Y}) : \hat{Y} \in \mathcal{L}(X)\}$. A solution (X,Y) is *optimal* if it minimizes $p(Y)$ over all feasible solutions. Note that determining whether (X,Y) is feasible is weakly NP-Hard.

Given that "the knapsack problem is believed to be one of the 'easier' NP-hard problems," [16] one may propose that BKP may also be one of the 'easier' Σ_2^p-hard problems. While this may indeed be the case, unlike KP, which admits a pseudopolynomial time algorithm, BKP remains NP-complete when the input is described in unary and thus has no pseudopolynomial time algorithm unless P = NP [1]. In addition, BKP is a Σ_2^p-complete problem, which means it cannot even be modelled as an IP with polynomially many variables and constraints, unless the polynomial hierarchy collapses. A recent positive theoretical result for BKP is a polynomial-time approximation scheme [3].

This theoretical hardness seemed to have been confirmed by the struggle of computational approaches to solve small instances. Until recently, proposed algorithms – either generic BIP algorithms [6,8,19] or more specific algorithms for BKP (or slight generalizations of it) [2,9,13] – were only able to solve instances with at most 55 items. A breakthrough result came in a paper by Della Croce and Scatamacchia [4], that proposed a BKP-specific algorithm (henceforth referred to as *DCS*) which was able to solve instances containing up to 500 items.

It is worth noting that all papers prior to DCS only consider instances which were generated in an *uncorrelated* fashion, meaning that weights and profits were chosen uniformly at random with no correlation between the values. The DCS algorithm is able to solve uncorrelated instances with 500 items in less than a minute, but its performance drops significantly even for weakly correlated instances, and most strongly correlated instances remain unsolved after an hour of computing time. These results seem to mimic what is known for KP: uncorrelated KP instances are some of the easiest types of instances to solve [16] and early KP algorithms such as `expknap` [15] could quickly solve uncorrelated instances but struggled with strongly correlated ones.

A common aspect among all methods in the literature is that they rely fundamentally on MIP solvers. In this paper, we present a simple combinatorial branch-and-bound algorithm for solving BKP. Our algorithm improves on the

performance of the DCS algorithm for 94% of instances, even achieving a speedup of orders of magnitude in many cases. Furthermore, our algorithm appears to be largely impervious to correlation: it solves strongly correlated instances with ease, only significantly slowing down when the lower-level weights equal the profits (i.e., the subset sum case). In Sect. 2, we describe our algorithm. Our algorithm relies fundamentally on a new strong lower bound computed by dynamic programming which we present in Sect. 3. Section 4 details our computational experiments. We conclude in Sect. 5 with some directions for future research. We note that some proofs were omitted for brevity.

2 A Combinatorial Algorithm for BKP

In this section we describe our exact solution method for BKP. At a high level, the algorithm is essentially just standard depth-first branch-and-bound. Our strong lower bound, defined later in Sect. 3, is essential for reducing the search space. To begin formalizing this, we first define the notion of a subproblem.

Definition 1. *A subproblem* (X, i) *consists of some* $i \in \{1, \ldots, n+1\}$ *and set of items* $X \subseteq \{1, \ldots i-1\}$ *such that* $X \in \mathcal{U}$*.*

Note that this definition depends on the ordering of the items, which throughout the paper we assume to be such that $\frac{p_1}{w_1^L} \geq \frac{p_2}{w_2^L} \geq \cdots \geq \frac{p_n}{w_n^L}$ with ties broken by placing items with larger p_i first. These subproblems will form the nodes of the branch-and-bound tree; $(\emptyset, 1)$ is the root node, and for every $X \in \mathcal{U}$, $(X, n+1)$ is a leaf. Every non-leaf subproblem (X, i) has the child $(X, i+1)$, which represents omitting item i from the upper-level solution. Non-leaf subproblems (X, i) with $X \cup \{i\} \in \mathcal{U}$ have an additional child $(X \cup \{i\}, i+1)$ which represents including item i in the upper-level solution.

The algorithm simply starts at the root and traverses the subproblems in a depth-first manner, preferring the child $(X \cup \{i\}, i+1)$ if it exists because it is more likely to lead to a good solution. Every time the search reaches a leaf $(X, n+1)$, we solve the knapsack problem $\max\{p(Y) : Y \in \mathcal{L}(X)\}$ to get a feasible solution, and updating the incumbent if appropriate.

2.1 The Bound Test

At each node (X, i) of the branch-and-bound, we find a lower bound on the optimal value of that subproblem by the use of a *bound test* algorithm, which tests lower bounds against a known incumbent solution value z^*.

The lower bound used to prune a subproblem is computed in three steps: (1) we solve a knapsack problem on items $\{1, \ldots, i-1\} \setminus X$, (2) we compute a lower bound for BKP restricted to items $\{i, \ldots, n\}$, and (3) we combine (1) and (2) into a lower bound for the descendants of (X, i).

For step (1), we define a function $K(\bar{X}, c)$, which, for a given $\bar{X} \subseteq \{1, \ldots, n\}$ and $c \geq 0$, returns the optimal value of the knapsack problem with weights w^L, profits p, and capacity c, under the restriction that items in $\bar{X}$ cannot be used:

$$K(\bar{X}, c) = \max\left\{p(Y) : Y \subseteq \{1, \ldots, n\} \setminus \bar{X} \text{ and } w^L(Y) \leq c\right\}.$$

For step (2), we need a function $\omega(i, c^U, c^L)$ which is a lower bound on BKP but with upper-level capacity c^U, lower-level capacity c^L, and restricted to items $\{i, \ldots, n\}$. So, formally, ω must satisfy

$$\omega(i, c^U, c^L) \leq \min\{K(X' \cup \{1, \ldots, i-1\}, c^L) : X' \subseteq \{i, \ldots, n\},\, w^U(X') \leq c^U\}.$$

We will define precisely what ω is in Sect. 3; for now, we only need to know that it has this property. We now prove the following lemma, which describes how to achieve step (3).

Lemma 1. *Let (X, i) be a subproblem. For all $c \in \{0, \ldots, C^L\}$,*

$$K(X \cup \{i, \ldots, n\}, c) + \omega\left(i, C^U - w^U(X), C^L - c\right)$$
$$\leq \min\left\{p(\bar{Y}) : (\bar{X}, \bar{Y}) \textit{ is feasible for BKP and } \bar{X} \cap \{1, \ldots, i-1\} = X\right\}.$$

Proof. First, note that for any $X' \subseteq \{i, \ldots, n\}$,

$$K(X \cup \{i, \ldots, n\}, c) + K(X' \cup \{1, \ldots, i-1\}, C^L - c) \leq K(X \cup X', C^L).$$

Thus, if we let $\chi' := \{X' \subseteq \{i, \ldots, n\} : w^U(X') \leq C^U - w^U(X)\}$ and take the minimum with respect to χ' we get

$$K(X \cup \{i, \ldots, n\}, c) + \omega(i, C^U - w^U(X), C^L - c) \leq \min\{K(X \cup X', C^L) : X' \in \chi'\}.$$

and now just note that this last term is equal to

$$\min\left\{p(\bar{Y}) : (\bar{X}, \bar{Y}) \text{ is feasible for BKP and } \bar{X} \cap \{1, \ldots, i-1\} = X\right\}. \quad \square$$

From this, it follows that for any $c \in \{0, \ldots, C^L\}$, if we have

$$K(X \cup \{i, \ldots, n\}, c) + \omega\left(i, C^U - w^U(X), C^L - c\right) \geq z^*$$

then we can prune subproblem (X, i). We also note that the lower bound still remains valid if we replace $K(X \cup \{i, \ldots, n\}, c)$ by a feasible solution to that problem. This is what is done in the function `BoundTest` in Algorithm 1 whose correctness is established by the following lemma.

Lemma 2. *If* `BoundTest(`X`,` i`)` *returns true, then subproblem (X, i) can be pruned.*

We end with an important consideration regarding the efficient implementation of Algorithm 1. The greedy part of the bound test (Lines 2 to 4) appears to require time $O(n)$. However, considering how we choose to branch, the values w_g and p_g can be computed in time $O(1)$ given their values for the parent subproblem. To determine $K(X \cup \{i, \ldots, n\}, c)$, we use the standard dynamic program (DP) for knapsack. However, for each bound test, it is only necessary to compute a single row of a DP table (i.e., fill in all C^L capacity values for the row associated with item i) from the row computed in the parent subproblem. By doing this, the entire `BoundTest` function will run in time $O(C^L)$. Furthermore, when the branch-and-bound reaches a leaf, the knapsack solution needed to update the upper bound will already have been found by the bound test.

Algorithm 1: Returns true if the subproblem (X, i) can be pruned.

```
  Precondition: z* is the value of the best incumbent solution
1 function BoundTest(X, i)
2     w_g, p_g ← 0;
3     for j = 1, ..., i − 1 do
4         if j ∉ X and w_g + w_j^L ≤ C^L then w_g ← w_g + w_j^L, p_g ← p_g + p_j;
5     if p_g + ω(i, C^U − w^U(X), C^L − w_g) ≥ z* then return true;
6     for c = 0, ..., C^L do
7         if K(X ∪ {i, ..., n}, c) + ω(i, C^U − w^U(X), C^L − c) ≥ z* then return
           true;
8     return false;
```

2.2 Computing Initial Bounds

In our algorithm, a strong initial upper bound z^* can help decrease the size of the search tree. For this we use a simple heuristic we call `GreedyHeuristic`. `GreedyHeuristic` works in two steps. First, an upper level set $\bar{X}$ is picked by solving $\max\{p(X) : X \in \mathcal{U}\}$. Then the lower level solution $\bar{Y}$ is picked by solving $\max\{p(Y) : Y \in \mathcal{L}(\bar{X})\}$. We say `GreedyHeuristic` returns $(\bar{X}, \bar{Y}, \bar{z})$, where $\bar{z}$ is the objective value of the solution $(\bar{X}, \bar{Y})$. We now establish a case in which `GreedyHeuristic` actually returns an optimal solution.

Lemma 3. `GreedyHeuristic()` *returns an optimal solution if there exists an optimal solution* (X, Y) *for BKP where* $Y = \{1, \ldots, n\} \setminus X$.

The proof is skipped for brevity, but we remark that previous work has noted that BKP is very easily solved for such instances [2,4]. Following [4], we use an LP to detect some cases where the `GreedyHeuristic` is optimal. The below formulation $LB(i)$ is a simplified version of the LP in [4].

$$
\begin{aligned}
LB(i) = \min \quad & \textstyle\sum_{j=1}^{i-1} p_j(1 - x_j) \\
\text{such that} \quad & \textstyle\sum_{j=1}^{i-1} w_j^U x_j \leq C^U \\
& C^L - w_i^L + 1 \leq \textstyle\sum_{j=1}^{i-1} w_j^L (1 - x_j) \leq C^L \\
& 0 \leq x \leq 1, \; x \in \mathbb{R}^{i-1}
\end{aligned}
$$

We define $LB(i) = \infty$ if the LP is infeasible. This LP is used by the following lemma. We skip the proof for brevity.

Lemma 4. *Suppose* `GreedyHeuristic()` *returns* $(\bar{X}, \bar{Y}, \bar{z})$. *If* $\bar{z} \leq \min\{LB(c) : 1 \leq c \leq n\}$ *then* $(\bar{X}, \bar{Y})$ *is optimal for BKP.*

Before starting our branch-and-bound algorithm, we run `GreedyHeuristic` and check if it is optimal using Lemma 4. This enables us to quickly solve trivial instances without needing to run our main algorithm.

3 Lower Bound

In this section we define the lower bound $\omega(i, c^U, c^L)$ that we use in our algorithm. Recall that $\omega(i, c^U, c^L)$ must lower bound the restriction of BKP where we can only use items $\{i, \ldots, n\}$, have upper-level capacity c^U, and lower-level capacity c^L. Our lower bound is based on dynamic programming (DP), which computes $\omega(i, c^U, c^L)$ for all parameter values with time and space complexity $O(nC^U C^L)$.

The main idea for the lower bound is to obtain good feasible solutions for the lower-level problem. Perhaps the most obvious way is to assume that the lower-level problem finds a greedy solution. It is not hard to see why this is a lower bound: a greedy lower-level solution will always achieve profit at most that of an optimal lower-level solution. We can compute this lower bound with the following recursively-defined DP algorithm:

$$\omega_g(i, c^U, c^L) =$$
$$\begin{cases} \infty & \text{if } c^U < 0, \\ -\infty & \text{if } c^L < 0, \\ 0 & \text{if } c^U \geq 0, c^L \geq 0 \text{ and } i > n, \\ \omega_g(i+1, c^U, c^L) & \text{if } c^U \geq 0,\ w_i^L > c^L \text{ and } i \leq n. \\ \min \left\{ \begin{array}{l} \omega_g(i+1, c^U - w_i^U, c^L), \\ \omega_g(i+1, c^U, c^L - w_i^L) + p_i \end{array} \right\} & \text{if } c^U \geq 0,\ w_i^L \leq c^L \text{ and } i \leq n. \end{cases}$$

The first three expressions are to take care of trivial cases. The fourth case skips any item which cannot fit in the lower-level knapsack, as it would be pointless for the upper level to take such an item. The fifth case picks the worse (i.e., better for the upper level) out of the two possible greedy solutions from the two children nodes of subproblem (X, i).

This lower bound already has very good performance in practice. However, we can do better by making a deceptively simple modification: giving the lower level the option to ignore an item. This modification produces our strong DP lower bound, ω, which is equal to ω_g in the first three cases, but instead of the last two cases we get:

$$\omega(i, c^U, c^L) = \min \left\{ \begin{array}{l} \omega(i+1, c^U - w_i^U, c^L), \\ \max \left\{ \begin{array}{l} \omega(i+1, c^U, c^L - w_i^L) + p_i, \\ \omega(i+1, c^U, c^L) \end{array} \right\} \end{array} \right\} \begin{array}{c} \text{if } c^U \geq 0,\ c^L \geq 0 \\ \text{and } i \leq n. \end{array}$$

It is not a hard exercise to show that $\omega_g(i, c^U, c^L) \leq \omega(i, c^U, c^L)$ for all $1 \leq i \leq n$, $0 \leq c^U \leq C^U$ and $0 \leq c^L \leq C^L$. Extrapolating our intuition about ω_g, formulation ω appears to actually find optimal lower-level solutions, so one might guess that $\omega(1, C^U, C^L)$ is actually optimal for BKP, if it weren't that this is impossible unless P = NP [1]. The subtlety is that by giving the lower level a choice of whether to take an item, we have also given the upper level the

power to react to that choice. Specifically, the upper level choice of whether to take item i can depend on how much capacity the lower level has used on items $\{1, \ldots, i-1\}$. Evidently, this is not permitted by the definition of BKP, which dictates that the upper level solution is completely decided prior to choosing the lower level solution. However, our experiments show that this actually gives the upper level an extremely small amount of additional power in practice.

The lower bound ω may also be interpreted as a relaxation from a 2-round game to a $2n$-round game. This may seem to be making the problem more difficult, but each round is greatly simplified, so the problem becomes easier to solve. This $2n$-round game is as follows. In round $2i-1$, the leader (the upper level player) decides whether to include the item i. In round $2i$, the follower (the lower level player) responds to the leader's decision: if item i is still available, then the follower decides whether to include item i. The score of the game is simply the total profit of all items chosen by the follower. It is straightforward to see that the minimax value of this game (i.e., the score given that both players follow an optimal strategy) is equal to $\omega(1, C^U, C^L)$.

We now show formally that $\omega(1, C^U, C^L)$ lower bounds the optimal objective value of BKP. To this end we define ω_X, a modified version of ω where instead of the minimization in the case where $c^U \geq 0$, $c^L \geq 0$ and $i \leq n$, the choice is made depending on whether $i \in X$ for some given set X. $\omega_X(i, c^U, c^L)$ is equal to ω_g in the first three cases, but replaces the last two cases with:

$$\omega_X(i, c^U, c^L) = \begin{cases} \omega_X(i+1, c^U - w_i^U, c^L) & \text{if } c^U \geq 0, c^L \geq 0, i \leq n \text{ and } i \in X, \\ \max \left\{ \begin{array}{l} \omega_X(i+1, c^U, c^L - w_i^L) + p_i, \\ \omega_X(i+1, c^U, c^L) \end{array} \right\} & \text{if } c^U \geq 0, c^L \geq 0, i \leq n \text{ and } i \notin X. \end{cases}$$

With this simple modification, we claim that $\omega_X(1, C^U, C^L) = \max\{p(Y) : Y \in \mathcal{L}(X)\}$ (and similarly for other i, c^U, and c^L). To formalize this, we show that ω_X and K (as defined in Sect. 2.1) are equivalent in the following sense.

Lemma 5. *For all* $1 \leq i \leq n$, $X \subseteq \{i, \ldots, n\}$, $c^U \geq w^U(X)$ *and* $c^L \geq 0$,

$$\omega_X(i, c^U, c^L) = K(X \cup \{1, \ldots, i-1\}, c^L).$$

Proof. Given that $c^U \geq w^U(X)$, the case $c^U < 0$ can not occur in the expansion of $\omega_X(i, c^U, c^L)$, so $\omega_X(i, c^U, c^L) = \omega_X(i, \infty, c^L)$. Consider the 0-1 knapsack problem with profits p' and weights w' formed by taking $p' = p$ and $w' = w^L$ except with $p'_j = w'_j = 0$ for items $j \in X$. We can then simplify the definition of $\omega_X(i, \infty, c^L)$ by using p' and w' to effectively skip items in X:

$$\omega_X(i, \infty, c^L) = \begin{cases} -\infty & \text{if } c^L < 0, \\ 0 & \text{if } c^L \geq 0 \text{ and } i > n, \\ \max \left\{ \begin{array}{l} \omega_X(i+1, \infty, c^L - w'_i) + p'_i, \\ \omega_X(i+1, \infty, c^L) \end{array} \right\} & \text{if } c^L \geq 0 \text{ and } i \leq n. \end{cases}$$

The recursive definition of $\omega_X(i, \infty, c^L)$ above describes the standard DP algorithm for 0-1 knapsack with capacity c^L, profits p' and weights w' but restricted to items $\{i, \ldots, n\}$; this is the same problem which is solved by $K(X \cup \{1, \ldots, i-1\}, c^L)$. □

We now establish that $\omega(i, c^U, c^L)$ is a lower bound as desired.

Theorem 1. *For all* $1 \leq i \leq n$*,* $c^U \geq 0$ *and* $c^L \geq 0$*,*

$$\omega(i, c^U, c^L) \leq \min_{X \subseteq \{i, \ldots, n\} : w^U(X) \leq c^U} K(X \cup \{1, \ldots, i-1\}, c^L).$$

Proof. By definition, $\omega_X(i, c^U, c^L) = \infty$ if $w^U(X) > c^U$, so

$$\begin{aligned} \min_{X \subseteq \{i, \ldots, n\}} \omega_X(i, c^U, c^L) &= \min_{X \subseteq \{i, \ldots, n\} : w^U(X) \leq c^U} \omega_X(i, c^U, c^L) \\ &= \min_{X \subseteq \{i, \ldots, n\} : w^U(X) \leq c^U} K(X \cup \{1, \ldots, i-1\}, c^L). \end{aligned}$$

where the last equality follows from Lemma 5. Therefore, it suffices to show that $\omega(i, c^U, c^L) \leq \min_{X \subseteq \{i, \ldots, n\}} \omega_X(i, c^U, c^L)$. The proof is by induction on i from $n+1$ to 1. Let $c^U \geq 0$ and $c^L \geq 0$ be arbitrary. Our inductive hypothesis is that $\omega(i, c^U, c^L) \leq \min_{X \subseteq \{i, \ldots, n\}} \omega_X(i, c^U, c^L)$. For the base case, where $i = n+1$, by definition we have $\omega(i, c^U, c^L) = \omega_X(i, c^U, c^L) = 0$ for any $X \subseteq \{i, \ldots, n\} = \emptyset$. Now we prove the inductive case. Let $1 \leq i \leq n$ be arbitrary and assume that the inductive hypothesis holds for $i+1$, with every $c^U \geq 0$ and $c^L \geq 0$. We present only the case where $w_i^U \leq c^U$ and $w_i^L \leq c^L$. The remaining cases (where $w_i^U > c^U$ or $w_i^L > c^L$) are just simpler versions of this.

$$\begin{aligned} \omega(i, c^U, c^L) &= \min \left\{ \begin{array}{l} \omega(i+1, c^U - w_i^U, c^L), \\ \max \left\{ \omega(i+1, c^U, c^L - w_i^L) + p_i, \omega(i+1, c^U, c^L) \right\} \end{array} \right\} \\ &\leq \min \left\{ \begin{array}{l} \min_{X \subseteq \{i+1, \ldots, n\}} \omega_X(i+1, c^U - w_i^U, c^L), \\ \max \left\{ \begin{array}{l} \min_{X \subseteq \{i+1, \ldots, n\}} \omega_X(i+1, c^U, c^L - w_i^L) + p_i, \\ \min_{X \subseteq \{i+1, \ldots, n\}} \omega_X(i+1, c^U, c^L) \end{array} \right\} \end{array} \right\} \\ &\leq \min \left\{ \begin{array}{l} \min_{X \subseteq \{i+1, \ldots, n\}} \omega_X(i+1, c^U - w_i^U, c^L), \\ \min_{X \subseteq \{i+1, \ldots, n\}} \max \left\{ \begin{array}{l} \omega_X(i+1, c^U, c^L - w_i^L) + p_i, \\ \omega_X(i+1, c^U, c^L) \end{array} \right\} \end{array} \right\} \\ &= \min_{X \subseteq \{i+1, \ldots, n\}} \min \left\{ \begin{array}{l} \omega_X(i+1, c^U - w_i^U, c^L), \\ \max \left\{ \begin{array}{l} \omega_X(i+1, c^U, c^L - w_i^L) + p_i, \\ \omega_X(i+1, c^U, c^L) \end{array} \right\} \end{array} \right\} \\ &= \min_{X \subseteq \{i, \ldots, n\}} \omega_X(i, c^U, c^L). \end{aligned}$$

□

Note that in particular, this implies that $\omega(1, C^U, C^L) \leq \min_{X \in \mathcal{U}} K(X, C^L) = \min_{X \in \mathcal{U}} \max_{Y \in \mathcal{L}(X)} p(Y)$, i.e., $\omega(1, C^U, C^L)$ is a lower bound for BKP.

We end this section with a simple observation. The approach we derived for our problem was based on obtaining good feasible solutions to the lower problem. Now, if the lower problem is already NP-hard, one may ask how useful can an approximate solution to the lower level be. For this, we consider a very generic problem:

$$z^* := \min_{x \in \mathcal{U}} \max_{y \in \mathcal{L}(x)} c(x, y) \tag{1}$$

For each $x \in \mathcal{U}$, assume there exists $y \in \mathcal{L}(x)$ that maximizes the inner problem. Let $y^*(x)$ be such a maximizer of $c(x, y)$ for $y \in \mathcal{L}(x)$. The following lemma then shows that if we can solve the problem with an approximate lower level, instead of an exact one, we get an approximate solution to (1).

Lemma 6. *Suppose we have a function $f(x)$ such that for all $x \in \mathcal{U}$:*

- *$f(x) \in \mathcal{L}(x)$, and*
- *$c(x, f(x)) \leq c(x, y^*(x)) \leq \alpha c(x, f(x))$, for some $\alpha \geq 1$.*

Let $\tilde{x} \in \arg\min_{x \in \mathcal{U}} c(x, f(x))$. Then $c(\tilde{x}, y^*(\tilde{x})) \leq \alpha z^*$.

Proof. Let $(x^*, y^*(x^*))$ be the optimal solution to (1). Then

$$\frac{1}{\alpha} c(\tilde{x}, y^*(\tilde{x})) \leq c(\tilde{x}, f(\tilde{x})) \leq c(x^*, f(x^*)) \leq c(x^*, y^*(x^*)) = z^*. \qquad \square$$

While this does not immediately give an approximation algorithm for the problem, we believe it may be useful to simplify some Σ_2^p-hard bilevel interdiction problems and, for that reason, we include this lemma in this work. Note that an analogous result can be also derived for a max – min problem.

4 Computational Results

In this section, we perform computational experiments to compare our algorithm (*Comb*) with the method from [4] (*DCS*). Given that the superiority of the DCS algorithm over other approaches has been well demonstrated we do not compare our algorithm directly to the prior works [2,6,8,9,18,19].

4.1 Implementation

We were unable to obtain either source code or a binary from the authors of [4], so we reimplemented their algorithm. We use Gurobi 9.5 instead of CPLEX 12.9, and obviously run it on a different machine, so an exact replication of their results is nearly impossible. Nonetheless, we found our reimplementation produces results very similar to what is reported in [4], and even solves three additional instances which were not solved in [4]. Therefore, we believe that any comparison with our version of the DCS algorithm is reasonably fair.

Both Comb and DCS were run using 16 threads. However, not all parts of the algorithms were parallelized. Specifically, in the DCS implementation, the only part which is parallelized is the MIP solver. In the implementation of Comb, we only parallelized two parts: the computation of the lower bound ω and the computation of the initial lower bound $\min\{LB(c) : 1 \leq c \leq n\}$.

Our code is implemented in C++ and relies on OpenMP 4.5 for parallelism, Gurobi 9.5 for solving MIPs, and the implementation of the `combo` knapsack algorithm [14] from [11]. The code was executed on a Linux machine with four 16-core Intel Xeon Gold 6142 CPUs @ 2.60 GHz and 256 GB of RAM. All code and instances are available at https://github.com/nwoeanhinnogaehr/bkpsolver.

4.2 Instances

Our test set contains all instances described in the literature [2,4,6,10,19] and 1660 new instances which we generated. The first 1500 were generated as follows. For each $n \in \{10, 25, 50, 10^2, 10^3, 10^4\}$, INS $\in \{0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5\}$ and $R \in \{10,\ 25, 50, 100, 1000\}$, we generated five instances according to five different methods, which we call classes 1-5. All weights and profits were selected uniformly at random in the range $[1, R]$, but for some of the five classes, we equated w^L, w^U or p with each other:

1. w^L, w^U and p are independent (uncorrelated)
2. $w^L = p$ but w^U is independent (lower subset-sum)
3. $w^U = p$ but w^L is independent (upper subset-sum)
4. $w^L = w^U = p$ (both subset-sum)
5. $w^L = w^U$ but p is independent (equal weights)

The capacities are chosen as follows. Let $C^L = \lceil \text{INS}/11 \cdot \sum_i w_i^L \rceil$ and choose C^U uniformly at random in the range $[C^L - 10, C^L + 10]$. If there is any item with $w_i^L < C^L$ or $w_i^U < C^U$, then we increase the appropriate capacity so that this is not the case. This is the same way that the capacities are selected in [2,4,10], except that we exclude instances that would almost certainly be solved by the initial bound test and we include half integral values of INS. Note that the easiest and hardest instances reported in the literature were uncorrelated and lower subset-sum, respectively [4]. Hence, we expect these new instances to capture both best-case and worst-case behavior from the solvers.

The remaining 160 instances were intended to test the case where the capacity is very large but the number of items is small. These instances were generated following the same scheme as above except that we only generated uncorrelated instances, and we chose $n \in \{5, 10, 20, 30\}$ and $R \in \{10^3, 10^4, 10^5, 10^6\}$. To the best of our knowledge, instances with such large capacity have not been evaluated previously.

4.3 Results

Our results on instances from the literature are summarized in Table 1. To best match the test environment used for the original DCS implementation, we ran

Table 1. Results for instances from the literature, grouped by instance type.

Group	Num	DCS				Comb			
		Opt	Best	Avg	Max	Opt	Best	Avg	Max
uncorrelated	940	940	66	2.32	15.73	940	874	0.31	6.48
weak correlated	50	50	0	13.49	72.64	50	50	0.26	3.54
strong correlated	50	41	0	689.58	3,600	50	50	0.34	3.98
inverse strong corr.	50	38	0	919.91	3,600	50	50	1.07	34.69
almost strong corr.	50	40	0	815.4	3,600	50	50	0.24	3.16
subset-sum	50	35	0	1,087.18	3,600	42	42	586.29	3,600
even-odd subset-sum	50	36	0	1,033.98	3,600	42	42	581.42	3,600
even-odd strong corr.	50	41	0	747.12	3,600	50	50	0.61	17.21
similar weight uncorr.	50	50	0	22.89	79.85	50	50	0.05	0.08

the tests with a time limit of 1 h, and used the same parameters for the DCS algorithm as reported by the authors [4]. For each instance group and each solver, we reported the number of instances solved to optimality (column Opt), the number of instances on which the solver took strictly less time than the other solver (column Best), the average wall-clock running time in seconds (column Avg) and the maximum wall-clock running time in seconds (column Max). Note that measuring wall-clock time as opposed to CPU time only disadvantages our algorithm, if anything, because the DCS implementation utilizes all 16 threads for a large proportion of the time due to parallelization within Gurobi, whereas the same is not true for our algorithm. Overall, our solver had better performance on 1258 of the 1340 instances (about 94%), achieving about 4.5 times better performance on average, and solving 53 of the 69 instances which our DCS implementation did not (the original DCS implementation did not solve 72 instances). These results demonstrate the remarkable advantage that Comb has on hard instances. DCS struggles with all instances involving strong or subset-sum correlation, but Comb only significantly slows down for subset-sum instances.

In Fig. 1, we plot a performance profile for instances from the literature comparing the DCS algorithm to some variants of our algorithm. This type of graph plots, for each instance, the ratio of each algorithm's performance to the performance of the best algorithm for that instance. The instances are sorted by difficulty. Note that instances which timed out are counted as 3600 s seconds. For a comprehensive introduction to performance profiles, see [7]. The two variants of Comb included are Comb-weak, which uses the lower bound ω_g instead of ω, and Comb-greedy, which uses a greedy lower bound test, i.e., where Lines 6 to 7 are omitted from Algorithm 1. The graph indicates that while Comb does better with more threads and the main variant is best, the single-threaded version and the variants still outperform 16-thread DCS. Although it is not depicted in the performance profile, we also tested variants with different item orderings, and found the one described in Sect. 2 to be the best. This is somewhat expected as this is the same ordering that gives rise to the greedy algorithm for 0-1 knapsack.

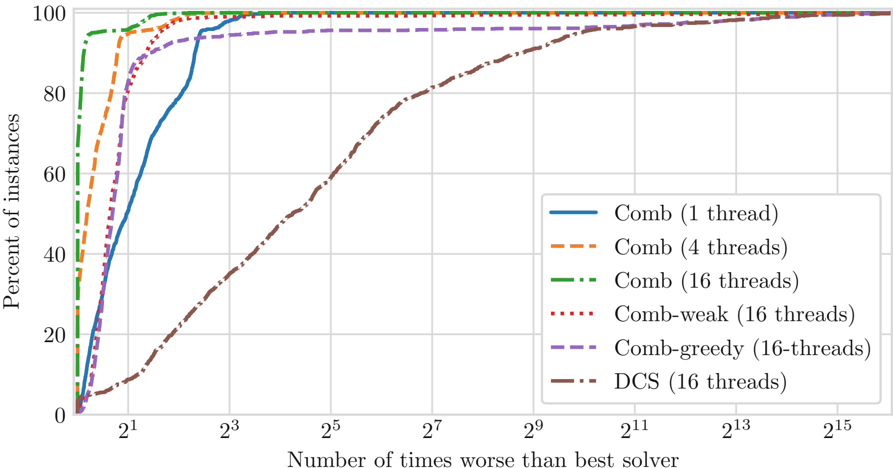

Fig. 1. Performance profile for all instances from the literature.

We now turn our attention to the new instances. Due to the large number of new instances and high difficulty, we used a reduced time limit of 15 min (900 s) in order to complete the testing in a timely fashion. For these tests we use the same DCS parameters used by the DCS authors for testing their own instances [4].

The results for the new instances are summarized in Tables 2 and 3. Note that there are 300 instances of each group, but Table 2 only instances for which our test machine had enough memory to store the DP table used in Comb. The performance of DCS is reported on the remaining instances in Table 3. We can see that Comb offers a significant speed improvement for all classes, and although both solvers are roughly equally capable of solving instances in the uncorrelated, upper subset-sum, equal weights, and large capacity classes, Comb solved 122 more of the instances in the lower subset-sum and both subset-sum classes than DCS. Extrapolating from the results in Table 2, we suspect that given sufficient memory, our solver would be able to solve many more of these instances with better performance than DCS.

In Tables 5 and 6, we report some statistics collected during the tests of our algorithm. In these tables, RootTime is the average number of (wall clock) seconds required to perform the initial bound test and compute the DP table, OptTime is the average number of seconds that branch-and-bound takes to find an optimal solution (excluding RootTime), ProofTime is the average number of seconds needed to prove optimality after an optimal solution is found, Nodes is the average number of nodes searched by branch-and-bound, and RootGap% is $100 \cdot (\bar{z} - \omega(1, C^U, C^L))/\bar{z}$ where $\bar{z}$ is the value of the solution returned by `GreedyHeuristic`. These tables only consider instances which fit in memory and did not time out, as some of the columns are undefined otherwise. Considering all

Table 2. Summary of results for new instances, grouped by class.

Group	Num	DCS				Comb			
		Opt	Best	Avg	Max	Opt	Best	Avg	Max
uncorrelated	243	241	3	12.42	900	243	240	0.96	24.02
lower subset-sum	256	173	0	320.36	900	236	236	73.62	900
upper subset-sum	235	235	5	2.7	89.25	235	230	0.79	21.08
both subset-sum	235	130	0	423.83	900	189	189	184.9	900
equal weights	236	236	9	2.2	120.55	236	227	0.7	18.38
large capacity	92	90	1	38.78	900	92	91	2.31	22.72

Table 3. Summary of results for DCS on new instances which our solver could not fit in memory, grouped by class.

Group	Num	Opt	Avg	Max
uncorrelated	57	41	487.85	900
lower subset-sum	44	1	886.75	900
upper subset-sum	65	59	283.34	900
both subset-sum	65	0	900	900
equal weights	64	62	175.92	900
large capacity	68	5	864.33	900

Table 4. An instance with n items, $C^U = n-1$ and $C^L = n$ that has optimal objective value $n-1$ but $\omega(1, C^U, C^L) = 1$.

item no.	p	w^U	w^L
1	1	1	1
⋮	⋮	⋮	⋮
n-1	1	1	1
n	n-1	n-1	n

instances which fit in memory, the average root gap is 3.22%, and the maximum is 57.89%.

Evidently, the root gap is typically very small and in fact $\omega(1, C^U, C^L) \geq 0.5\,\text{OPT}$ in all tested instances. However, the worst case approximation factor of ω is actually unbounded. Table 4 describes a family of instances with n items for which $\text{OPT} = (n-1)\omega(1, C^U, C^L)$. Despite this, branch-and-bound is able to solve these instances using only $O(n)$ nodes. On the other hand, it is interesting to note that the subset-sum instances typically have very small root gaps, but solving them to optimality is evidently very hard.

Evidently, the main disadvantage of our algorithm is its high memory usage. In our solver, we use a few simple tricks to reduce memory usage slightly: when possible, we store the DP table entries as 16-bit integers, and we avoid computing table entries for capacity values which cannot be seen in any feasible solution. Other optimizations to reduce memory usage are certainly possible as well, such as a DP-with-lists type approach, but we have not implemented this.

Table 5. Statistics from our solver, for instances from the literature.

Group	RootTime	OptTime	ProofTime	NumNodes	RootGap%
uncorrelated	0.3	0.03	0.02	12,053.23	5.4
weak correlated	0.25	0.04	0.03	7,130.98	2.04
strong correlated	0.26	0.03	0.11	571,473.6	2.98
inverse strong corr.	0.37	0.04	0.73	4,751,888.08	0.39
almost strong corr.	0.24	0.03	0.03	735.76	2.99
subset-sum	0.08	0.05	12.22	104,067,718.79	0.02
even-odd subset-sum	0.07	0.04	6.42	61,246,957.64	0.02
even-odd strong corr.	0.26	0.03	0.37	2,551,213.42	2.95
similar weight uncorr.	0.05	0.05	0.05	0	0

Table 6. Statistics from our solver, for new instances.

Group	RootTime	OptTime	ProofTime	Nodes	RootGap%
uncorrelated	0.86	0.27	0.17	73,616.37	9.86
lower subset-sum	0.95	0.8	3.43	11,293,085.12	1.27
upper subset-sum	0.79	0	0	182.99	5.75
both subset-sum	0.16	3.02	7.68	117,664,085.82	3.44
equal weights	0.7	0.02	0.02	14.65	1.64
large capacity	2.31	0	0	29.82	10.15

5 Conclusion

We presented a new combinatorial algorithm for solving BKP that is on average 4.5 times better, and achieves up to 3 orders of magnitude improvement in runtime over the performance of the previous state-of-the-art algorithm, DCS. The only disadvantage of our algorithm that we identified in computational testing is the high memory usage. Because of this, if memory is limited and time is not a concern, it may be a better idea to use DCS. However, if there is any correlation between the lower-level weights and profits, DCS is unlikely to solve the instance in any reasonable amount of time, so it is preferable to use our algorithm on a machine with a large amount of memory, and/or to use additional implementation tricks to reduce the memory usage.

For future work, it would be of interest to investigate whether our lower bound can be strengthened further (say, to an $O(1)$-approximation). We expect that it would be straightforward to generalize this work to the multidimensional variant of BKP (i.e., where there are multiple knapsack constrains at each level), although the issues with high memory usage would likely become worse in this setting. It may also be straightforward to apply this technique to covering interdiction problems. Beyond this, we suspect that a similar lower bound and search algorithm can be used to efficiently solve a variety of interdiction problems.

References

1. Caprara, A., Carvalho, M., Lodi, A., Woeginger, G.J.: A study on the computational complexity of the bilevel knapsack problem. SIAM J. Optim. **24**(2), 823–838 (2014)
2. Caprara, A., Carvalho, M., Lodi, A., Woeginger, G.J.: Bilevel knapsack with interdiction constraints. INFORMS J. Comput. **28**(2), 319–333 (2016)
3. Chen, L., Wu, X., Zhang, G.: Approximation algorithms for interdiction problem with packing constraints. arXiv preprint arXiv:2204.11106 (2022)
4. Della Croce, F., Scatamacchia, R.: An exact approach for the bilevel knapsack problem with interdiction constraints and extensions. Math. Program. **183**(1), 249–281 (2020)
5. Dempe, S.: Bilevel optimization: theory, algorithms, applications and a bibliography. In: Dempe, S., Zemkoho, A. (eds.) Bilevel Optimization. SOIA, vol. 161, pp. 581–672. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-52119-6_20
6. DeNegre, S.: Interdiction and discrete bilevel linear programming, Ph. D. thesis, Lehigh University (2011)
7. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Math. Program. **91**(2), 201–213 (2002)
8. Fischetti, M., Ljubić, I., Monaci, M., Sinnl, M.: A new general-purpose algorithm for mixed-integer bilevel linear programs. Oper. Res. **65**(6), 1615–1637 (2017)
9. Fischetti, M., Ljubic, I., Monaci, M., Sinnl, M.: Interdiction games and monotonicity, with application to knapsack problems. INFORMS J. Comput. **31**, 390–410 (2019)
10. Fischetti, M., Monaci, M., Sinnl, M.: A dynamic reformulation heuristic for generalized interdiction problems. Eur. J. Oper. Res. **267**, 40–51 (2018)
11. Fontan, F.: Knapsack solver (Github repository). https://github.com/fontanf/knapsacksolver (2017)
12. Kleinert, T., Labbé, M., Ljubić, I., Schmidt, M.: A survey on mixed-integer programming techniques in bilevel optimization. EURO J. Comput. Optimiz. **9**, 100007 (2021)
13. Lozano, L., Bergman, D., Cire, A.A.: Constrained shortest-path reformulations for discrete bilevel and robust optimization. arXiv preprint arXiv:2206.12962 (2022)
14. Martello, S., Pisinger, D., Toth, P.: Dynamic programming and strong bounds for the 0–1 knapsack problem. Manage. Sci. **45**(3), 414–424 (1999)
15. Pisinger, D.: An expanding-core algorithm for the exact 0–1 knapsack problem. Eur. J. Oper. Res. **87**(1), 175–187 (1995)
16. Pisinger, D.: Where are the hard knapsack problems? Comput. Oper. Res. **32**, 2271–2284 (2005)
17. Smith, J.C., Song, Y.: A survey of network interdiction models and algorithms. Eur. J. Oper. Res. **283**(3), 797–811 (2020)
18. Tahernejad, S., Ralphs, T.K., DeNegre, S.T.: A branch-and-cut algorithm for mixed integer bilevel linear optimization problems and its implementation. Math. Program. Comput. **12**(4), 529–568 (2020)
19. Tang, Y., Richard, J.P.P., Smith, J.C.: A class of algorithms for mixed-integer bilevel min-max optimization. J. Global Optim. **66**, 225–262 (2016)