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. Previous exact methods for solving this problem make use of mixed-integer linear programming solvers. 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. Our algorithm is simple: a basic implementation takes less than 200 lines of code. More drastic performance improvements are seen for more challenging instances: on 20% of instances, our algorithm is at least 64 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 relaxes the problem from bilevel to $2n$-level by ordering the items and solving an online version of the problem. 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.

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 [KLLS21, Dem20, SS20]. In the BIP model, there are two IPs, called the upper level and lower level, 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. Due to this optimality constraint, most BIPs cannot be modeled as IPs without using an exponential number of constraints and/or variables. As such, BIPs are generally considered very hard to solve, compared to IPs: the problem of solving a general BIP is Σ_2^p-complete, whereas the problem of solving a general IP is only NP-complete (equivalently, Σ_1^p-complete) [DeN11, CCLW14].

In the literature, sometimes the upper level is referred to as the *leader's* problem and the lower level as the *follower's* problem. These terms come from the theory of Stackelberg games, from which bilevel programming originated [VS52]. For consistency, in this paper we exclusively use the terms upper and lower.

Recent advancements in BIP solvers have made it possible to solve larger, more complex BIPs, but it is recognized that there is still room for improvement [SS20, KLLS21]. In particular, problem-specific algorithms far outpace general BIP algorithms. Due to the difficulty of creating strong general-purpose BIP solvers, there is interest in further developing problem-specific algorithms as a means to develop insight into the general case. In this paper we focus on the bilevel knapsack problem with interdiction constraints (BKP), which was introduced by DeNegre in 2011 [DeN11]. This problem is a natural extension of the 0-1 knapsack problem (KP) to the bilevel setting.

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 [SS20]).

BKP is considered to be a fundamental Σ_2^p-complete bilevel problem. In fact, it is currently the only Σ_2^p-complete problem which is known to admit a polynomial-time approximation scheme [CWZ22]. Given that "the knapsack problem is believed to be one of the 'easier' NP-hard problems," [Pis05] one may propose (particularly after seeing our results section) that BKP is one of the 'easier' Σ_2^p-hard problems. However, unlike KP, which admits a psuedopolynomial time algorithm, BKP remains NP-complete when the input is described in unary and thus has no psuedopolynomial time algorithm unless P = NP [CCLW14]. For these reasons, it is of great interest to develop fast algorithms for solving BKP so that the nature of BIPs and the class of Σ_2^p-complete problems can be better understood.

Intuitively, BKP can be thought of as a two player game where each player has a knapsack which they can pack some items into. Each item has some associated profit value. The players are selecting from the same collection of items, but the items may have different weight depending on which player picks them, and the two knapsacks may also have different capacity. The objective is for the upper-level player (first player) to pack some items into their knapsack so that the maximum profit that the lower-level player (second player) can achieve using the remaining items is minimized. The upper-level player has no regard for the profit of the items they choose; they are only interested in minimizing profit for the lower-level player.

Formally, the problem is defined as follows. A problem instance consists of 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 $C^L \in \mathbb{Z}_{\geq 0}$. Notice that we assume that the profits p and upper level weights w^U are non-zero, but the lower level weights w^L are permitted to be zero. We define $\mathcal{U}$ to contain all feasible sets of items for the upper-level knapsack, and given some $X \in \mathcal{U}$, we define $\mathcal{L}(X)$ to contain all feasible sets of items (excluding items in X) for the lower-level knapsack. Finally, we define some 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 pairs. Note that given X, determining whether $Y \in \text{argmax}\{p(\hat{Y}) : \hat{Y} \in \mathcal{L}(X)\}$ amounts to solving a 0-1 knapsack problem, so determining feasibility is already weakly NP-Hard.

Along with introducing the problem, DeNegre designed a general purpose branch-and-cut framework for mixed-integer bilevel programs and evaluated the approach on BKP instances with up to 15 items [DeN11]. Since then, solution methods have been gradually improved over a series of papers. In 2016, the first BKP-specific algorithm, the CCLW algorithm, was developed [CCLW16]. This algorithm was able to solve instances with up to 50 items, but could not solve some instances with 55 items within an hour. Later that same year, a new general purpose branch-and-cut algorithm was introduced in [TRS16] which was able to solve instances with up to 30 items. The following year, a paper [FLMS17] introduced a new general-purpose algorithm called `MIX++` which achieves better performance compared to [TRS16]. The 2019 paper [FLMS19] develops a branch-and-cut algorithm for a class of interdiction problems which generalizes BKP. Their approach is able to solve the 55-item instances which [CCLW16] could not solve. A very recent paper [LBC22] considers reformulating the lower-level problem as a shortest path problem by constructing a binary decision diagram, so that the bilevel problem can be written as a single MIP. This approach is tested on BKP instances with up to 50 items and has good performance compared to other general-purpose methods, especially when the item weights and capacities are small.

All of these papers 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. It is well known that uncorrelated KP instances are some of the easiest types of instances to solve [Pis05]. Similarly, current algorithms are able to solve uncorrelated BKP instances much faster than correlated instances. A heuristic approach proposed in [FMS18] was the first to consider a variety of correlated BKP instances. In particular, they consider instances where the profit p_i is very close to the lower-level weight w_i^L for all i.

In 2018, Della Croce and Scatamacchia published a BKP-specific algorithm which they tested on all known instances as well as newly generated instances containing up to 500 items [DCS20]. Their algorithm

Algorithm 1: Main branch and bound algorithm for BKP.

Precondition: (X^*, Y^*) is a feasible solution and $z^* = p(Y^*)$ (see Section 2.2).

```
1 function BranchAndBound(X, i)
2     if i = n + 1 then
3         Y ← argmax{p(Y) : Y ∈ L(X)};
4         if p(Y) < z* then X* ← X, Y* ← Y, z* ← p(Y);
5         return;
6     if BoundTest(X, i) then return;
7     if X ∪ {i} ∈ U then BranchAndBound(X ∪ {i}, i + 1);
8     BranchAndBound(X, i + 1);
```

is significantly stronger than the previous approaches. We refer to this algorithm as the DCS algorithm. While the DCS algorithm is able to solve uncorrelated instances with 500 items in less than a minute, the performance drops significantly even for weakly correlated instances, and most strongly correlated instances remain unsolved after an hour of compute time. This performance is reminiscent of earlier KP algorithms such as `expknap` [Pis95], which could quickly solve uncorrelated instances but struggled with strongly correlated instances.

All BKP solution methods discussed so far 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 95% of instances, with a speedup by 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 Section 2, we describe our algorithm. Our algorithm relies fundamentally on a new strong lower bound computed by dynamic programming which we present in Section 3. Section 4 details our computational experiments, demonstrating that our algorithm outperforms the previous state-of-the-art approach, the DCS algorithm. We conclude in Section 5 with some directions for future research.

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 Section 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. If this solution improves upon our best solution seen so far, then we replace the best solution with it. At each subproblem we perform a bound test using the best solution in combination with a lower bound to determine if the subproblem can be pruned. This algorithm is stated formally as the `BranchAndBound` function in Algorithm 1.

2.1 The bound test

The purpose of the bound test is to prune subproblems from the search that cannot lead to an improved solution. As with any branch and bound algorithm, the bound test is key to achieving good performance. In this section we assume that z^* is an upper bound on the objective value. We will see how the initial upper bound is computed in Section 2.2. At a high level, the bound test works as follows. Given a subproblem (X, i) we compute a lower bound on the profit achievable by leaves in the subtree rooted at (X, i). If this lower bound is at least z^*, then we know that no descendant of (X, i) could possibly improve upon z^*, so we prune (X, i).

The lower bound 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 $K(X, c)$ as the optimal value of the knapsack problem with weights w^L, profits p, and capacity c, but restricted to not use items in X:

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

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 Section 3; for now, we only need to know that it has this property. We now prove the following lemma, which describes how to combine steps (1) and (2) into (3), a lower bound for the descendants of a subproblem (X, i).

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

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

Proof. Let (X, i) be a subproblem and let $c \in \{0, \ldots, C^L\}$. By definition,

$$\begin{aligned} &K(X \cup \{i, \ldots, n\}, c) + \omega(i, C^U - w^U(X), C^L - c) \\ \leq\, &K(X \cup \{i, \ldots, n\}, c) + \min\{K(X' \cup \{1, \ldots, i-1\}, C^L - c) : X' \subseteq \{i, \ldots, n\},\, w^U(X') \leq C^U - w^U(X)\} \\ \leq\, &\min\{K(X \cup X', C^L) : X' \subseteq \{i, \ldots, n\},\, w^U(X') \leq C^U - w^U(X)\} \\ =\, &\min\{K(X \cup X', C^L) : X' \subseteq \{i, \ldots, n\},\, w^U(X' \cup X) \leq C^U\} \\ =\, &\min\{p(Y') : (X', Y') \text{ is feasible for BKP and } X' \cap \{1, \ldots, i-1\} = X\}. \end{aligned}$$

□

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

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

then we can prune subproblem (X, i) because it would be impossible for any leaf which is a descendant of (X, i) to correspond with a feasible solution of objective value less than z^*. Because performing this test is computationally expensive, in our bound test we first try a variant of the test in which a greedy solution is used in place of $K\left(X \cup \{i, \ldots, n\}, c\right)$. We define the test formally by the function `BoundTest` in Algorithm 2 and prove correctness in the following lemma.

Lemma 2. *If* `BoundTest(`X`,` i`)` *returns true, then subproblem (X, i) can be pruned in Algorithm 1.*

Proof. Consider the greedy algorithm on Lines 1 to 4 of Algorithm 2. This algorithm finds a feasible solution with weight w_g and profit p_g for the knapsack problem with weights w^L, profits p, and capacity C^L but restricted to items in $\{1, \ldots, i-1\} \setminus X$. Therefore, $p_g \leq KP\left(X \cup \{i, \ldots, n\}, w_g\right)$, so by Lemma 1 the function only returns true on Line 4 if subproblem (X, i) can be pruned. The correctness of Line 7 follows immediately from Lemma 1. □

Algorithm 2: Returns true if the subproblem (X, i) can be pruned.

```
  Precondition: z* is an upper bound
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;
```

This concludes the main description of the bound test, but there is an important consideration regarding the efficient implementation of Algorithm 2. We may assume that ω is precomputed and takes constant time to query. The greedy part of the bound test (Lines 1 to 4) appears to require time $O(n)$. However, considering the recursive structure of `BranchAndBound`, the values w_g and p_g can be computed in time $O(1)$ given their values for the parent subproblem. In a similar manner, we do not need to solve an entire knapsack problem every time `BoundTest` is called to determine $K(X \cup \{i, \ldots, n\}, c)$. Instead, it is only necessary to compute a single row of a knapsack dynamic programming 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 in Line 3 of Algorithm 1 will already have been found by the bound test; all that is needed is to recover it by traversing the dynamic programming table.

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 the *greedy heuristic*, which is defined as the function `GreedyHeuristic` in Algorithm 3. This heuristic finds a feasible solution for BKP: suppose (X^*, Y^*, z^*) is the output of `GreedyHeuristic`; then, since $X^* \in \mathcal{U}$ and $Y^* \in \operatorname{argmax}\{p(Y) : Y \in \mathcal{L}(X^*)\}$, (X^*, Y^*) is feasible for BKP with objective value z^*. Hence, z^* is an upper bound. This heuristic can be computed by solving two knapsack problems, which can be done efficiently with an algorithm such as `combo` [MPT99]. We now establish a case in which the greedy heuristic actually returns an optimal solution.

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

Proof. We denote $\overline{X} := \{1, \ldots, n\} \setminus X$. Let (X, Y) be an optimal solution for BKP where $Y = \overline{X}$. We claim that $X \in \operatorname{argmax}\{p(X) : X \in \mathcal{U}\}$. If not, then there is some $X' \in \mathcal{U}$ with $p(X') > p(X)$. Let $Y' \in \arg\max \{p(Y) : Y \in \mathcal{L}(X')\}$. Then

$$p(Y') \leq p(\overline{X'}) < p(\overline{X}) = p(Y)$$

so (X', Y') contradicts the optimality of (X, Y). Now, if `GreedyHeuristic()` returns a solution (X', Y') which is suboptimal for BKP, we reach a contradiction:

$$p(\overline{X}) = p(Y) < p(Y') \leq p(\overline{X'}) = p(\overline{X})$$

□

In previous work, it has been noted that BKP is very easily solved for such instances [DCS20, CCLW16]. Intuitively, this case occurs when the lower-level capacity is large enough that the lower level can pack all items given any upper-level solution, or when the upper-level capacity is large enough that the upper level can pack all items. As we will see in Section 3, the time and space complexity of our strong lower bound ω is $O(nC^UC^L)$, so when the capacities are large we would like to avoid computing it at all if possible. Therefore, to detect this case, we use a simpler lower bound: a polynomially sized LP formulation suggested

Algorithm 3: Greedy heuristic.

```
1 function GreedyHeuristic()
2     X* ← argmax {p(X) : X ∈ 𝒰};            // use a knapsack algorithm such as combo [MPT99]
3     Y* ← argmax {p(Y) : Y ∈ ℒ(X*)};
4     return (X*, Y*, p(Y*));
```

Algorithm 4: Solves BKP.

```
1 global X*, Y*, z* ← GreedyHeuristic();
2 if z* ≤ min {LB(c) : 1 ≤ c ≤ n} then output (X*, Y*, z*);     // initial bound test (optional)
3 BranchAndBound(∅, 1);
4 output (X*, Y*, z*);
```

by Della Croce and Scatamacchia [DCS20]. In their paper, they define a formulation denoted $CRIT_1^{LP}(c)$, which we have simplified into the below formulation $LB(c)$ that is sufficient for our purposes. We remark that the use of this LP means our algorithm is not purely combinatorial. However, this LP is an optional part of our algorithm which only provides a performance increase in some easy cases.

$$\begin{aligned} LB(c) = \min \quad & \textstyle\sum_{i=1}^{c-1} p_i(1-x_i) \\ \text{such that} \quad & \textstyle\sum_{i=1}^{c-1} w_i^U x_i \leq C^U \\ & C^L - w_c^L + 1 \leq \textstyle\sum_{i=1}^{c-1} w_i^L (1-x_i) \leq C^L \\ & 0 \leq x \leq 1 \\ & x \in \mathbb{R}^{c-1} \end{aligned}$$

If the LP is infeasible for some c, we define $LB(c) = \infty$. The following lemma enables us to detect some cases where the greedy heuristic is optimal.

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

Proof. By Lemma 3, if an optimal solution (X^*, Y^*) exists where $Y^* = \{1, \dots, n\} \setminus X^*$, then (X, Y) is optimal. Otherwise, there must be an optimal solution (X^*, Y^*) for which $c^* = \min\{c : \sum_{i \notin X^* : i \leq c} w_i^L > C^L\}$ exists. Let $x \in \{0,1\}^{c^*-1}$ be such that $x_i = 1$ if and only if $i \in X^*$ for $1 \leq i \leq c^* - 1$. Then x is feasible for $LB(c^*)$, so $LB(c^*) \leq \sum_{i=1}^{c^*-1} p_i(1-x_i) \leq p(Y^*)$ and therefore $\min\{LB(c) : 1 \leq c \leq n\}$ is a lower bound. Since z is an upper bound, either (X, Y) is optimal because $z = \min\{LB(c) : 1 \leq c \leq n\}$, or $z > \min\{LB(c) : 1 \leq c \leq n\}$. □

To achieve good performance from this lower bound, it is necessary to sort the items beforehand as described at the beginning of Section 2. Furthermore, note that it is not necessary to compute $LB(c)$ for every choice of c as some options can be easily ruled out. For brevity, we omit this minor detail here and refer readers to the original paper [DCS20].

Putting everything together, we arrive at Algorithm 4.

Theorem 1. *Algorithm 4 correctly solves BKP.*

Proof. By Lemma 4, Algorithm 4 only outputs (X^*, Y^*, z^*) on Line 2 if (X^*, Y^*, z^*) is optimal. Otherwise, since `BranchAndBound` enumerates all upper-level solutions except those pruned by `BoundTest` (which we proved correctness for in Lemma 2), it will either reach some optimal leaf and update (X^*, Y^*, z^*) accordingly, or prove that the (X^*, Y^*, z^*) returned by `GreedyHeuristic` actually was optimal (despite that it could not be proved by Line 2). □

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. In our branch-and-bound algorithm, this lower bound may be queried for a very large number of different parameter values (possibly most values of $1 \leq i \leq n$, $0 \leq c^U \leq C^U$ and $0 \leq c^L \leq C^L$). The lower bound from [DCS20] is strong but requires solving as many as n IPs to find the lower bound for just a single set of parameters (i, c^U, c^L), so it would be very inefficient to use this lower bound for ω. We instead introduce a new lower bound based on dynamic programming (DP), which computes $\omega(i, c^U, c^L)$ for all parameter values with time and space complexity $O(nC^UC^L)$.

Perhaps the most obvious way to compute a lower bound suiting our needs is to solve a modified version of BKP which uses a greedy solution to the lower-level knapsack problem, i.e., the lower level processes items from 1 to n and always takes an item when there is enough remaining capacity to do so. 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:

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

Here, the first case, $c^U < 0$, is used to force the min expression in the fourth case not to pick a choice which goes above the upper-level capacity. The second case, where $c^U \geq 0, c^L \geq 0$ and $i > n$, simply terminates the recursion when there are no items left to process. The third 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 fourth case considers the decision of whether the upper level should take item i, blocking the lower level from taking it, or whether the lower level should take it (the lower level has no option to ignore the item when following the greedy algorithm).

This lower bound already has very good performance in practice, as demonstrated in Section 4.3 (see algorithm Comb-Weak). However, we can do better by making a deceptively simple modification: giving the lower level the option to ignore an item, while otherwise keeping the structure of the DP algorithm the same. This modification produces our strong DP lower bound, ω, described as follows:

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

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 [CCLW14]. 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, as we will see in Section 4, 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. Assume that an ordering of the items is fixed. The game starts at round 1. 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 in round $2i-1$: if the leader does not include item i, then the follower decides whether to include item i; if the leader does include item i, then the follower has no choice to make and the game progresses immediately to round $2i+1$. 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.

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

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 Section 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:

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

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 2. *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

$$\begin{aligned} \min_{X \subseteq \{i, \ldots, n\}} \omega_X(i, c^U, c^L) &= \min_{X \subseteq \{i, \ldots, n\} : w^U(X) \leq c^U} \omega_X(i, c^U, c^L) \\ &= \min_{X \subseteq \{i, \ldots, n\} : w^U(X) \leq c^U} K(X \cup \{1, \ldots, i-1\}, c^L). \qquad \text{(by Lemma 5)} \end{aligned}$$

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 conclude the proof with four cases, all of which follow simply from the definitions and inductive hypothesis.

Case 1: *$w_i^U > c^U$ and $w_i^L > c^L$.*

$$\omega(i, c^U, c^L) = \omega(i+1, c^U, c^L) \leq \min_{X \subseteq \{i+1,\ldots,n\}} \omega_X(i+1, c^U, c^L) = \min_{X \subseteq \{i,\ldots,n\}} \omega_X(i, c^U, c^L).$$

Case 2: *$w_i^U \leq c^U$ and $w_i^L > c^L$.*

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

Case 3: *$w_i^U > c^U$ and $w_i^L \leq c^L$.*

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

Case 4: *$w_i^U \leq c^U$ and $w_i^L \leq c^L$.*

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

□

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 would also like some idea of how far from optimal it can be in the worst case. The worst example we were able to find, described in Table 1, achieves a gap of 2, i.e., the optimal solution has objective value $2\omega(1, C^U, C^L)$. However, we do not have a proof that 2 is the worst-case gap. In any case, the gap is significantly better than 2 for all instances we tested computationally, as detailed in the next section.

item no.	p	w^U	w^L
1	1	1	1
2	1	1	1
3	2	2	3

Table 1: An instance with $C^U = 2$ and $C^L = 3$ that has optimal objective value 2 but $\omega(1, C^U, C^L) = 1$.

4 Computational results

In this section, we compare our algorithm to the method from [DCS20] using computational tests. We call our algorithm Comb (short for combinatorial) and refer to the algorithm from [DCS20] as DCS. We remark that there are many other published algorithms capable of solving BKP, e.g., [DeN11, CCLW16, TRS16, FLMS17, FLMS19, TRD20]. However, the superiority of the DCS algorithm has been well demonstrated by its authors, so we do not compare our algorithm directly to prior works.

4.1 Implementation

Our implementation of both algorithms is open-source (released under the MIT license) and is available at https://github.com/nwoeanhinnogaehr/bkpsolver. Scripts to run experiments from this section are included to ease reproducing our results. We used the C++ programming language and rely on OpenMP 4.5 for parallelism, Gurobi 9.5 for solving MIPs, and Florian Fontan's Knapsack Solver (https://github.com/fontanf/knapsacksolver) for its implementation of the `combo` algorithm. Our code was compiled with Clang 14 and run on a Linux machine with four 16-core Intel Xeon Gold 6142 CPUs @ 2.60GHz and 256GB of RAM. We limited the solvers to use at most 16 threads because the machine was shared with other users and we saw only marginal increases in performance when using more than 16 threads.

We were unable to obtain either source code or a binary from the authors of the DCS algorithm, but our reimplementation largely matches the performance reported in the original paper. Of the instances reported in the literature, our reimplementation solves three additional instances which were not solved by the original implementation and achieves very similar performance across the whole test set. Any differences are likely explained by the fact that the original implementation used CPLEX 12.9 and was run on an Intel i5 CPU @ 3.0 GHz. The original implementation was run using the default CPLEX parameters, but we used the Gurobi parameters `MIPFocus=2`, `Presolve=2`, `PreSparsify=1` and `Cuts=0` as we found them to increase performance. Given that the performance of our reimplementation was similar to the performance reported in their paper, and sometimes even better, we believe that any comparison with our version of the DCS algorithm is reasonably fair.

As mentioned, both algorithms 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. Since the algorithm spends almost all of the running time within the MIP solver, it is expected that parallelizing other parts of the algorithm would result in negligible speedup. In the implementation of our algorithm, we parallelized two parts: the computation of the lower bound ω (by filling in neighbouring elements of the DP table which do not depend on each other in parallel) and the computation of the initial lower bound $\min\{LB(c) : 1 \leq c \leq n\}$ (by computing $LB(c)$ for all values of c in parallel). For uncorrelated instances, computing the lower bound takes almost all of the running time, so they are effectively parallelized. However, for correlated instances most of the time is spent performing the branch-and-bound search. The branch-and-bound search could be parallelized to speed up this case, but we did not implement this optimization.

4.2 Instances

Our test set consists of the following groups of instances from the literature:

- **CCLW:** This group contains 50 instances from [CCLW16] with n ranging from 35 to 55. Weights and profits are uncorrelated and range between 1 and 100. The lower-level capacity C^L is set to some

fraction of the sum of the weights, and the upper-level capacity C^U is randomly chosen in the range $[C^L - 10, C^L + 10]$.
- **DCS:** This group contains 500 instances from [DCS20] with n ranging from 100 to 500. These instances are generated in the same way as the CCLW instances.
- **DeNegre:** This group contains 160 instances from [DeN11] with n ranging from 10 to 50. Note that the original paper only tested instances with n up to 15, but the instances available for download (from https://coral.ise.lehigh.edu/) contain up to 50 items. Weights and profits are uncorrelated and range from 1 to 1000. Capacities are relatively large, up to about 15000.
- **FMS:** This group contains 450 instances from [FMS18] with n ranging from 100 to 500. Upper-level weights are selected uniformly at random. Lower-level weights and profits are generated according to nine different classes with varying levels of correlation. For example, the subset-sum class has $p_i = w_i^L$ for all i, the strongly correlated class has $p_i = w_i^L + 10$ for all i, and the weakly correlated class has $p_i \in [w_i^L - 10, w_i^L + 10]$ selected uniformly at random for all i. For a full description of all instance classes, we refer the reader to [FMS18]. Capacities are selected the same way as in the CCLW instances. In our results we split these instances into two groups. FMS-easy contains three classes: uncorrelated, weakly correlated, and uncorrelated with similar weights. FMS-hard contains the remaining classes: variants on strongly correlated and subset sum.
- **TRS:** This group contains 180 instances from [TRS16] with n ranging from 15 to 30 in which all upper-level weights w_i^U are 1. Lower-level weights and profits are uncorrelated and range between 1 and 100. Lower-level capacity ranges up to about 700 and upper level capacity is at most 23.

In addition, we generated 1500 new instances. With these instances we intended to test some cases which had not been evaluated in the literature. We generated some very large instances (with up to 10000 items), instances with a smaller or larger weights than reported in the literature, and instances that have correlation between the upper-level weights and the lower-level weights or the profits. Specifically, 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 [CCLW16, DCS20, FMS18], except that we test a smaller range of capacities (excluding instances that would almost certainly be solved by the initial bound test described in Section 2.2) and we test more capacities in the range by including half integral values of INS. See Fig. 2 for the results of an experiment that further motivates choosing the capacities in this way.

Note that the easiest and hardest instances reported in the literature were uncorrelated and lower subset-sum, respectively [DCS20]. Hence, we expect our new instances to capture both best-case and worst-case behaviour from the solvers.

4.3 Results

Our results on instances from the literature are summarized in Table 2. To best match the test environment used for the original DCS implementation, we ran the tests with a time limit of 1 hour, and used the same parameters for the DCS algorithm as reported by the authors [DCS20]. 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).

Group	#Inst	DCS #Opt	DCS #Best	DCS Avg	DCS Max	Comb #Opt	Comb #Best	Comb Avg	Comb Max
All	1,340	1,271	55	200.49	3,600	1,324	1,269	44.11	3,600
CCLW	50	50	2	0.21	1.27	50	48	0.04	0.07
DCS	500	500	0	3.87	15.73	500	500	0.7	8.59
DeNegre	160	160	50	0.17	1.62	160	110	0.08	1.73
FMS-easy	150	150	0	13.35	79.85	150	150	0.38	7.1
FMS-hard	300	231	0	882.2	3,600	284	284	195.61	3,600
TRS	180	180	3	0.14	2.04	180	177	0.04	0.05

Table 2: Summary of results for all instances from the literature.

Class	DCS #Opt	DCS #Best	DCS Avg	DCS Max	Comb #Opt	Comb #Best	Comb Avg	Comb Max
uncorrelated	50	0	3.66	13.38	50	50	0.64	7.1
weak correlated	50	0	13.49	72.64	50	50	0.39	4.76
strong correlated*	41	0	689.58	3,600	50	50	0.46	5.02
inverse strong corr.*	38	0	919.91	3,600	50	50	1.17	31.11
almost strong corr.*	40	0	815.4	3,600	50	50	0.35	4.28
subset-sum*	35	0	1,087.18	3,600	42	42	588.57	3,600
even-odd subset-sum*	36	0	1,033.98	3,600	42	42	582.37	3,600
even-odd strong corr.*	41	0	747.12	3,600	50	50	0.73	17.06
similar weight uncorr.	50	0	22.89	79.85	50	50	0.12	0.35

Table 3: Summary of results for FMS instances. All classes contain 50 instances. Classes in FMS-hard are marked with a star (*).

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 the duration of the tests due to parallelization within Gurobi. Overall (see row All), our solver had better performance on 1269 of the 1340 instances (about 95%), achieving about 4.5 times better performance on average, and solving 53 of the 72 instances which DCS did not.

Note that on groups consisting of uncorrelated or weakly correlated instances (all except FMS-hard), the two solvers appear equally capable of finding optimal solutions, as long as slightly more time is given to DCS. However, the results for FMS-hard demonstrate that Comb is better at solving hard instances. We further examine this behavior in Table 3, where it can be seen that DCS struggles with all instances in the group FMS-hard, where as our algorithm only significantly slows down for subset-sum instances.

In Fig. 1, we plot a performance profile comparing the DCS algorithm to some variants of our algorithm using instances from the literature. 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. For example, the graph indicates that on about 80% of instances, the DCS algorithm is at most 2^6 times slower than the best algorithm for that instance, and that Comb is never worse than 4 times slower than any other algorithm. Note that instances which timed out are counted as 3600 seconds. For a comprehensive introduction to performance profiles, see [DM02]. We included several variants of our algorithm to demonstrate that the main variant (Comb) has the best performance. Comb-Greedy is the variant where only the greedy lower bound test is performed, i.e., Lines 6 to 7 are removed from Algorithm 2. Variant Comb-Weak uses the weaker lower bound ω_g described in Section 3. From the plot it is clear that Comb performs significantly better than DCS. Although Comb-Greedy is faster than Comb on a few easy instances, it appears to be worthwhile to do the more expensive bound test used in Comb (this may be difficult to see from the graph because Comb and Comb-Greedy are very similar in performance). Comb-Weak does not have any advantage over Comb as the lower bound is only marginally less expensive to compute but is much weaker. Yet, Comb-Weak still manages to perform better than DCS in most cases.

We now turn our attention to the new instances. Due to the large number of new instances and high

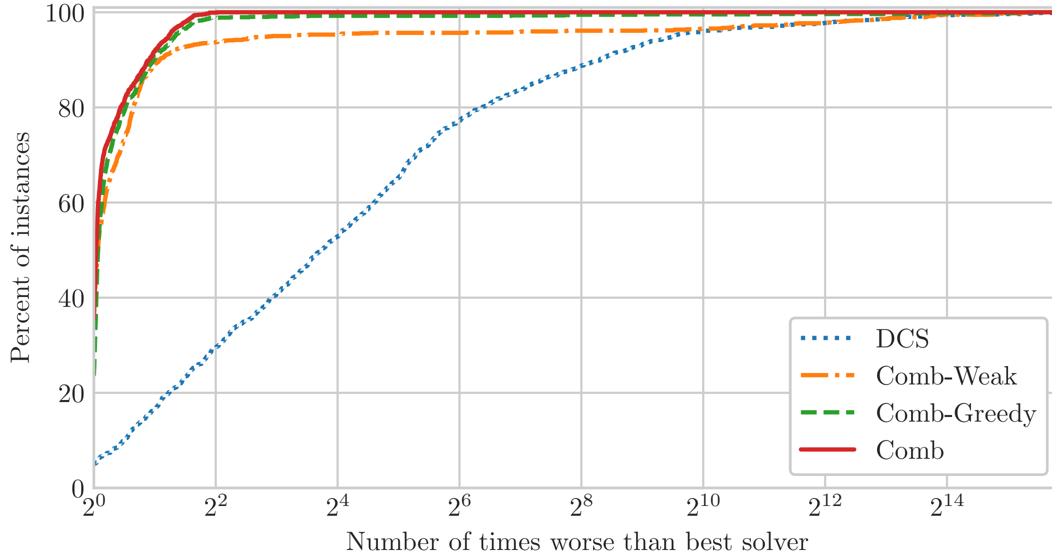

Figure 1: Performance profile for all instances from the literature.

difficulty, we used a reduced time limit of 15 minutes (900 seconds) in order to complete the testing in a timely fashion. For these tests we use the DCS parameters $\mu = 1000$, $\alpha = 2$, $\beta = 2$, $\gamma = 5$, and $\omega = 5$, which are the standard parameters used by the DCS authors for testing their own instances [DCS20]. For a few of the new instances, Gurobi can sometimes get stuck and be unable to solve some MIPs if cuts are disabled, so unlike the other tests, we enabled cuts (`Cuts=1`), even though this reduces speed slightly on the easier instances.

The results for the new instances are summarized in Tables 4 and 5. Table 4 considers only instances where our test machine had enough memory to attempt using our solver. We tested the remaining instances using only the DCS algorithm and summarized the results in Table 5. In both tables, we grouped instances by n in the upper half of the table and by class in the lower half. In Table 4, we can see that Comb offers a significant speed improvement for all n and all classes. While the speed of Comb evidently scales more slowly than DCS with n, it is somewhat unclear what happens at the largest two sizes of n because many instances do not fit in memory. In the grouping by instance class, we can see that both solvers are roughly equally capable of solving instances in the uncorrelated, upper subset-sum, and equal weights classes. However, Comb solved 122 more of the instances in the lower subset-sum and both subset-sum classes than DCS. Out of the 307 instances where there was insufficient memory to use our algorithm, the DCS algorithm was able to solve 171 of them, taking an average of 515.33 seconds per instance. Note that none of these 171 instances had any correlation between the lower-level weights and profits, which are classes where Comb has a distinct advantage. Extrapolating from the results in Table 4, we suspect that given sufficient memory, our solver would be able to solve many more of these instances with better performance than DCS.

Unexpectedly, the easiest and hardest types of instances were not uncorrelated and lower subset-sum, as suggested by prior work. Rather, the easiest classes are in fact *equal weights*, in which $w^U = w^L$ but p is chosen independently, and *upper subset-sum*, in which $w^U = p$ but w^L is chosen independently. The hardest class is *both subset-sum*, in which $w^U = w^L = p$; some instances from this class with only 25 items could not be solved by DCS. None of these three classes have been studied previously. When given an instance of the both subset-sum class (or lower subset-sum), our solver typically computes a very tight lower bound, almost always differing by less than 5% from the optimal solution (see Fig. 2), but due to the extremely large number of optimal and near-optimal solutions in these instances, the branch-and-bound search must recurse very deep before it is able to prune any branches. Hence, the algorithm ends up enumerating an exponential number of branch-and-bound nodes. This behavior is somewhat similar to the DCS algorithm: when given such an instance, it ends up enumerating an extremely large number of near-optimal solutions, each found individually by solving a MIP (to get an upper-level solution) and then solving a KP instance

n	#Inst	DCS #Opt	DCS #Best	DCS Avg	DCS Max	Comb #Opt	Comb #Best	Comb Avg	Comb Max
10	250	250	101	0.13	3.41	250	149	0.05	0.13
25	250	238	8	58.36	900	250	242	0.05	0.33
50	250	203	1	178.63	900	247	246	17.83	900
100	250	184	3	253.42	900	222	219	104.77	900
1000	167	109	12	302.26	900	136	124	169.82	900
10000	26	23	0	357.43	900	26	26	12.55	25.21
Class									
uncorrelated	241	239	25	12.37	900	241	216	0.97	25.21
lower subset-sum	256	174	13	318.09	900	237	224	70.2	900
upper subset-sum	232	232	31	2.58	89.25	232	201	0.8	18.29
both subset-sum	232	130	23	417.68	900	189	166	175.93	900
equal weights	232	232	33	2.12	120.55	232	199	0.67	14.97

Table 4: Summary of results for new instances, grouped by n (upper half) and by class (lower half). This table only includes instances which our solver could fit in memory; the remaining instances are summarized in Table 5.

n	#Inst	DCS #Opt	DCS Avg	DCS Max
1000	83	47	422.77	900
10000	224	124	550.41	900
Class				
both subset-sum	68	0	900	900
lower subset-sum	44	0	900	900
equal weights	68	66	165.98	900
upper subset-sum	68	62	271.35	900
uncorrelated	59	43	471.94	900

Table 5: Summary of results for DCS on new instances which our solver could not fit in memory, grouped by n (upper half) and by class (lower half).

(to get a lower-level solution).

Evidently, the main disadvantage of our algorithm is its high memory usage. A simple optimization to reduce the memory usage is to use 16-bit integers for storing the entries of the lower bound DP table instead of 32-bit. This is possible for all instances from the literature because the instances are sufficiently small that it is impossible to achieve profit larger than 2^{16}. However, for some of our new instances, a 32-bit table is required. Our implementation automatically detects whether a 16-bit table can be used and prefers that option if possible. Other optimizations to reduce memory usage are certainly possible, such as only storing part of the lower-bound table and recomputing it as needed, or compressing the table in memory. We decided not to pursue these ideas because they would constitute more of an engineering effort than a theoretical improvement to the algorithm.

In Fig. 2, we report the results of two additional experiments intended to illustrate some properties of our algorithm. The first, depicted in the left side of the figure, graphs the approximation ratio of our lower bound ω for instances from the literature as well as our new instances. The approximation ratio is defined as $\omega(1, C^U, C^L)/\text{OPT}$ where OPT is the optimal objective value of BKP. From the graph, we can see that we actually have $\omega(1, C^U, C^L) = \text{OPT}$ for around 85% of the instances. Moreover, except for two instances (one being the worst known example, described in Table 1), we always have $\omega(1, C^U, C^L)/\text{OPT} \geq 0.8$. In the second experiment, depicted on the right side of the figure, we plot the effect of the knapsack capacities on the running time. Our motivation for this experiment is twofold: it demonstrates what types of instances can be solved by the initial bound test, and it justifies our choice to pick similar upper-level and lower-level capacities

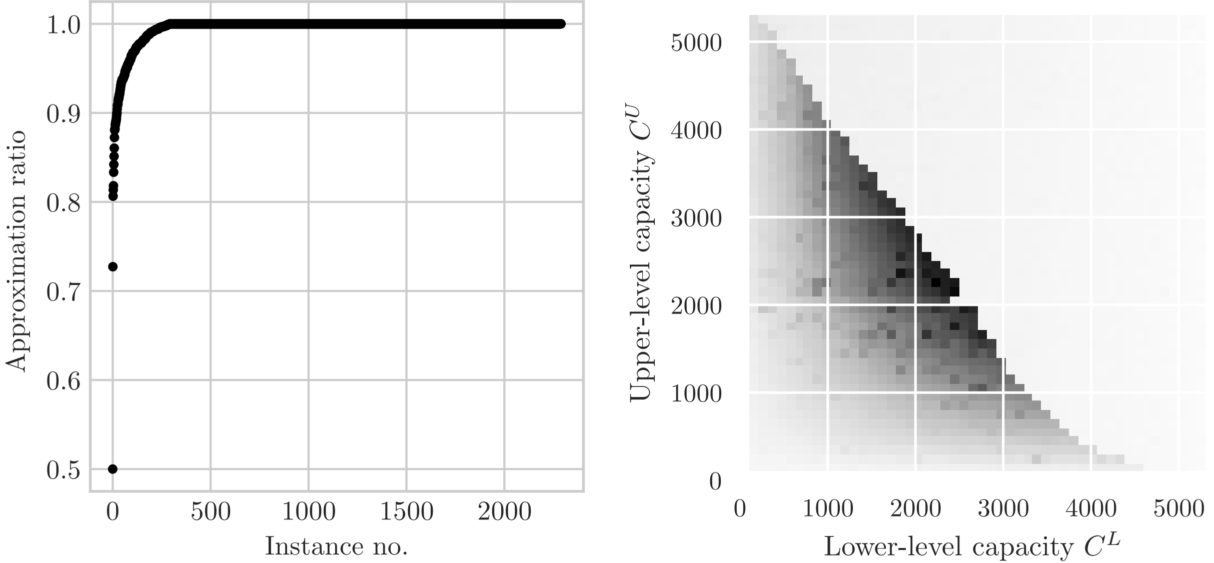

Figure 2: Left: scatter plot of the lower bound approximation ratio $\omega(1, C^U, C^L)/\mathrm{OPT}$ for all instances (sorted by approximation ratio). Right: running time of Comb (darker = longer time) as a function of C^U and C^L, for an uncorrelated instance with 100 items.

in our new instances. To create this figure, we generated an instance with 100 items, following the same generation scheme as the CCLW and DCS instances. We then solved the instance but with many different values of C^U and C^L. Although the figure only considers a single instance, the behavior is representative of the general case. In the figure, darker colors indicate a longer running time, with black indicating about 140 milliseconds and white indicating 0. We can see that there is a threshold reached when the capacities are sufficiently large where the initial bound test becomes able to solve the problem near-instantly (in less than 10 milliseconds). The most difficult cases have C^U and C^L very close to each other and as large as possible without the initial bound test being able to solve the problem.

5 Conclusion

We have described 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, for very large capacity uncorrelated instances, it is generally a better idea to use DCS. However, even for instances with very large capacity, if there is any correlation between the lower-level weights and profits, our results indicate that DCS is unlikely to solve the instance, so it is preferable to use our algorithm on a machine with a large amount of memory, or to use additional implementation tricks to reduce the memory usage (perhaps with a slight decrease in performance).

For future work, it would be of interest to prove a bound on the approximation factor of our strong lower bound, and to investigate whether it can be strengthened further. 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

[CCLW14] Alberto Caprara, Margarida Carvalho, Andrea Lodi, and Gerhard J Woeginger. A study on the computational complexity of the bilevel knapsack problem. *SIAM Journal on Optimization*, 24(2):823–838, 2014.

[CCLW16] Alberto Caprara, Margarida Carvalho, Andrea Lodi, and Gerhard J Woeginger. Bilevel knapsack with interdiction constraints. *INFORMS Journal on Computing*, 28(2):319–333, 2016.

[CWZ22] Lin Chen, Xiaoyu Wu, and Guochuan Zhang. Approximation algorithms for interdiction problem with packing constraints. *arXiv preprint arXiv:2204.11106*, 2022.

[DCS20] Federico Della Croce and Rosario Scatamacchia. An exact approach for the bilevel knapsack problem with interdiction constraints and extensions. *Mathematical Programming*, 183(1):249–281, 2020.

[Dem20] Stephan Dempe. Bilevel optimization: theory, algorithms, applications and a bibliography. In *Bilevel optimization*, pages 581–672. Springer, 2020.

[DeN11] Scott DeNegre. *Interdiction and discrete bilevel linear programming.* PhD thesis, Lehigh University, 2011.

[DM02] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[FLMS17] Matteo Fischetti, Ivana Ljubić, Michele Monaci, and Markus Sinnl. A new general-purpose algorithm for mixed-integer bilevel linear programs. *Operations Research*, 65(6):1615–1637, 2017.

[FLMS19] Matteo Fischetti, Ivana Ljubic, Michele Monaci, and Markus Sinnl. Interdiction games and monotonicity, with application to knapsack problems. *INFORMS Journal on Computing*, 31:390–410, 2019.

[FMS18] Matteo Fischetti, Michele Monaci, and Markus Sinnl. A dynamic reformulation heuristic for generalized interdiction problems. *European Journal of Operations Research*, 267:40–51, 2018.

[KLLS21] Thomas Kleinert, Martine Labbé, Ivana Ljubić, and Martin Schmidt. A survey on mixed-integer programming techniques in bilevel optimization. *EURO Journal on Computational Optimization*, 9:100007, 2021.

[LBC22] Leonardo Lozano, David Bergman, and Andre A Cire. Constrained shortest-path reformulations for discrete bilevel and robust optimization. *arXiv preprint arXiv:2206.12962*, 2022.

[MPT99] Silvano Martello, David Pisinger, and Paolo Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Management Science*, 45(3):414–424, 1999.

[Pis95] David Pisinger. An expanding-core algorithm for the exact 0–1 knapsack problem. *European Journal of Operational Research*, 87(1):175–187, 1995.

[Pis05] David Pisinger. Where are the hard knapsack problems? *Computational Operations Research*, 32:2271–2284, 2005.

[SS20] J Cole Smith and Yongjia Song. A survey of network interdiction models and algorithms. *European Journal of Operational Research*, 283(3):797–811, 2020.

[TRD20] Sahar Tahernejad, Ted K Ralphs, and Scott T DeNegre. A branch-and-cut algorithm for mixed integer bilevel linear optimization problems and its implementation. *Mathematical Programming Computation*, 12(4):529–568, 2020.

[TRS16] Yen Tang, Jean-Philippe P. Richard, and Jonathan Cole Smith. A class of algorithms for mixed-integer bilevel min–max optimization. *Journal of Global Optimization*, 66:225–262, 2016.

[VS52] Heinrich Von Stackelberg. *The theory of the market economy.* Oxford University Press, 1952.