The 0–1 Knapsack Problem

Alberto Santini

Fall 2024

Many examples in these lecture notes are adapted from popular books:

- Alexander Schrijver (1998). *Theory of linear and integer programming*. Wiley. ISBN: 0-471-98232-6.
- Vašek Chvátal (1983). *Linear Programming*. W.H. Freeman and Company. ISBN: 0-716-71195-8.
- Laurence Wolsey (2020). *Integer Programming*. 2nd Edition. Wiley. ISBN: 978-1-119-60653-6.
- Silvano Martello and Paolo Toth (1990). *Knapsack Problems: algorithms and computer implementations*. Wiley. ISBN: 978-0-471-92420-3.

The 0–1 Knapsack Problem (01-KP) is one of the most studied problems in Operational Research. It is easy to state and formulate, yet it can be challenging to solve computationally. It also has many practical applications and appears as a subproblem in other relevant problems. Finally, it has many variants and extensions, further increasing its role as a real-world modelling tool.

The 01-KP is usually introduced in a tongue-in-cheek way in the following setting. Imagine being a thief who is stealing from an apartment. You find n interesting objects, which you denote as $J = \{1, \dots, n\}$. Each object $j \in J$ has a given integer weight $w_j \in \mathbb{N}^+$ and yields an integer profit $p_j \in \mathbb{N}^+$ when sold on the black market. You can only carry objects whose total weight does not exceed $c \in \mathbb{N}^+$ units of weight; otherwise, your rucksack (or *knapsack*) will break. The 01-KP asks to select the subset of objects which yield the largest profit without exceeding the maximum weight c.

It is usually assumed that (i) each object fits in the knapsack individually, i.e., $w_j \leq c \; \forall j \in J$; (ii) taking *all* objects would be infeasible, i.e., $\sum_{j \in J} w_j > c$; (iii) objects are numbered according to their *density* (the ratio of profit to weight), i.e., $p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$.

1 A binary programming formulation

In this section, we present a binary programme that models the 01-KP. The model uses one variable per object and only has one constraint. This formulation is (apparently!) extremely simple and has been studied deeply for more than five decades.

Let $x_j \in \{0, 1\}$ be a binary variable defined for each object $j \in J$. We want x_j to be equal to one if object j should be packed in the knapsack and $x_j = 0$ otherwise. Then, a formulation for the 01-KP reads as follows.

$$\max \quad \sum_{j \in J} p_j x_j \tag{1a}$$

$$\text{subject to} \quad \sum_{j \in J} w_j x_j \tag{1b}$$

$$x_j \in \{0, 1\} \qquad \forall j \in J. \tag{1c}$$

The objective function (1a) maximises the profit of the packed objects. Constraint (1b) imposes that the sum of the weights of the packed objects does not exceed the knapsack's capacity.

We also define the bounded continuous relaxation of formulation (1a)–(1c). Differently from a standard continuous relaxation in which variable domain definition $x_j \geq 0$ would replace $x_j \in \{0, 1\}$, in the following *bounded* relaxation, we use definition $x_j \in [0, 1]$. This is a shorthand for using the standard $x_j \geq 0$ and adding constraint $x_j \leq 1$. This choice is motivated by the fact that, without this restriction, the relaxation of the 01-KP would be trivial. An optimal solution to the standard continuous relaxation would completely fill the knapsack with object 1 because it has the best profit-to-weight ratio by the density assumption ($p_1/w_1 \geq p_j/w_j$ for all $j \in J \setminus \{1\}$). The solution would then assign value c/w_1 to variable x_1 and value 0 to all other variables. Because, in virtually all practical cases, $c/w_1 > 1$, this solution would correspond to taking multiple copies of object 1. In the real world, our thief cannot clone objects; therefore, this relaxation would not be helpful.

Considering the following relaxation, on the contrary, ensures that each object is packed at most once.

$$\max \quad \sum_{j \in J} p_j x_j \tag{2a}$$

$$\text{subject to} \quad \sum_{j \in J} w_j x_j \tag{2b}$$

$$x_j \in [0, 1] \qquad \forall j \in J. \tag{2c}$$

Intuitively, in relaxation (2a)–(2c), the thief can pack a fraction of an object. Doing so, he will carry the corresponding fraction of the object's weight and gain the corresponding fraction of the object's profit.

As we will see in the rest of the lecture notes, this relaxation has a particular structure, which we can exploit when devising a branch-and-bound (BB) algorithm for the 01-KP. For the moment, we only remark that (2a)–(2c) is, indeed, a *relaxation* of (1a)–(1c), because the original variable domain $\{0, 1\}$ is contained in the relaxed one $[0, 1]$. As a consequence, the objective value z^*_{cont} of the optimal solution of (2a)–(2c) is an upper bound on the objective value z^* of the optimal solution of (1a)–(1c).

Exercise 1. You are managing the bid allocation of advertisement space during the mid-game break of the Uefa Champions League final. Management has allocated c seconds to show ads during the break. During the bidding period, n advertisers have presented their offers. Advertiser j's commercial lasts w_j seconds, and the advertiser will pay you p_j euros if you show their ad.

1. Write an integer programme to accept the subset of bids that yield the highest overall profit and ensure that the total length of the ads does not exceed the available time.
2. Some advertisers offer to pay extra money if you show their ad, and *do not* show their competitors'. Let $\mathcal{C}$ be a set of ordered pairs (i, j) of advertisers. Each pair $(i, j) \in \mathcal{C}$ has an associated extra profit π_{ij} that you earn if you decide to accept i's bid but refuse j's. Update your integer programme to take this new scenario into account.

2 A branch-and-bound algorithm for the 01-KP

Formulation (1a)–(1c) is a binary programme, i.e., all its variables are binary. Therefore, all branching constraints in a BB algorithm to solve this problem will be of type $x_j \leq 0$ or $x_j \geq 1$. Written more intuitively, these constraints state that either object j is excluded from the knapsack ($x_j \leq 0$, i.e., $x_j = 0$) or it is included ($x_j \geq 1$, i.e., $x_j = 1$).

Given the above observation, we can present a BB algorithm for the 01-KP more intuitively than a generic BB algorithm for an integer programme. In fact, we will use a BB tree in which each branching corresponds to either packing or excluding a single object. In particular, the decision on whether to pack object j will be taken at the j-th level of the tree.

2.1 Exploration strategy

Regarding the exploration strategy, we will always explore a node associated with a "pack" decision ($x_j = 1$) if such a node exists. Otherwise, we explore the last "do not pack" node created, i.e., one associated with a $x_j = 0$ decision. This rule ensures that only one "pack" node can be open at any time.

2.2 Obtaining dual bounds

A key component for the success of the algorithm we present is that the bounded continuous relaxation (2a)–(2c) has a particular structure that allows us to compute its optimal solution quickly, without having to solve a Linear Programme explicitly. This result is presented in the following theorem.

Theorem 1. *Recall that the objects are numbered by decreasing density. Let the* ***critical object*** *be the object with index s, where*

$$s = \min\Big\{i \in J \ : \ \sum_{j=1}^{i} w_j > c\Big\}. \tag{3}$$

In other words, s is the first object (in the given order) that does not fit in the knapsack. Remark that $2 \leq s \leq n$ because we assume each object fits individually in the knapsack and not all objects fit when taken together.

Then, the following is an optimal solution of (2a)–(2c)*:*

$$\begin{cases} x_j = 1 & \forall j \in J,\ j < s \\ x_s = \frac{c - \sum_{j \in J,\ j<s} w_j}{w_s} & \\ x_j = 0 & \forall j \in J,\ j > s. \end{cases} \tag{4}$$

The optimal solution packs all objects in order until they fit entirely in the knapsack. Then, it packs a fraction of the first object that does not fit (the critical object). The fraction is such that the capacity of the knapsack is used completely. No other object is packed.

The quantity $\bar{c} = c - \sum_{j \in J,\ j<s} w_j$ is called the ***residual capacity*** *of the knapsack after packing objects $\{1, \ldots, s-1\}$.*

Intuitively, if we change solution (4), we must remove at least part of an object in $\{1, \ldots, s\}$ and replace it with an object (or part of an object, or multiple parts of objects) in $\{s+1, \ldots, n\}$. However, since the density of objects $\{s+1, \ldots, n\}$ is not higher than that of objects $\{1, \ldots, s\}$,

such a change cannot increase the collected profit. As a consequence of Theorem 1, and considering that profits are integers, the objective value of the optimal solution of the bounded continuous relaxation is

$$\bar{z} = \left\lfloor \sum_{j \in S, j<s} p_j + \frac{\bar{c}}{w_s} p_s \right\rfloor. \tag{5}$$

Theorem 1 gives us a way to solve the bounded continuous relaxation of the 01-KP at the root node. Next, we show how to solve this relaxation at the other tree nodes, where a given number of constraints is active. Let $P \subseteq J$ be the set of objects j for which a constraint $x_j = 1$ is active, $\bar{P} \subseteq J$ the set of objects j for which a constraint $x_j = 0$ is active, and $F = J \setminus (P \cup \bar{P})$ the set of objects for which no constraint is active. We call the objects of P **fixed**, those of $\bar{P}$ **excluded**, and those of F **free**.

The fixed objects are forced to be packed in the knapsack. Therefore, we have to make no decision about them; at the same time, they will consume capacity $w_P = \sum_{j \in P} w_j$. We also have to make no further decision about the excluded objects, which do not occupy any capacity. At a given node, then, we should only decide what subset of the free objects to pack into the residual capacity $c - w_P$. We can solve a 01-KP with objects F and capacity $c - w_P$ to make this decision. Let $(x_j^*)_{j \in F}$ be an optimal solution to this problem. This solution can be extended into an optimal solution of the overall 01-KP with the additional constraints active at the given node by setting $x_j^* = 1$ for $j \in P$ and $x_j^* = 0$ for $j \in \bar{P}$.

Analogously, an optimal solution to the bounded continuous relaxation of the 01-KP at a given node is given by extending an optimal solution $(x_j^*)_{j \in F}$ of the bounded continuous relaxation optimal solution of a 01-KP with objects F and capacity $c - w_P$. Considering this observation, we can solve the bounded relaxation and obtain the corresponding dual (upper) bound at any tree node.

2.3 Branching strategy

Unlike a classic BB algorithm for an integer programme, we do not necessarily branch on fractional variables in this tailored BB algorithm for the 01-KP. Indeed, we mentioned earlier that the decision on packing or excluding object j is taken at the j-th level of the tree. This implies that we will branch on x_1 at the root node, x_2 at its two children nodes, etc.

This branching strategy saves us some computation compared to the classical one in which we would have to branch on the only possibly fractional variable associated with the critical object. If we branched on x_s, both children problems associated with $x_s = 0$ and $x_s = 1$ would be different from the parent problem (except in the exceptional case in which $x_s = 0$ because objects $\{1, \ldots, s-1\}$ perfectly fill the capacity). With our branching scheme, instead, we must not perform any computation in the $x_j = 1$ branch as long as $x_j^* = 1$ in the father node: the optimal solution of the child node associated with $x_j = 1$ is the same as the father node solution.

2.4 An example

Consider a 01-KP instance with $n = 5$ objects and capacity $c = 10$. Let the profits and weights be

$$(p_j)_{j \in J} = (12, 12, 7, 6, 2)$$
$$(w_j)_{j \in J} = (4, 5, 3, 3, 2).$$

We will solve this 01-KP instance using the BB algorithm described in this section.

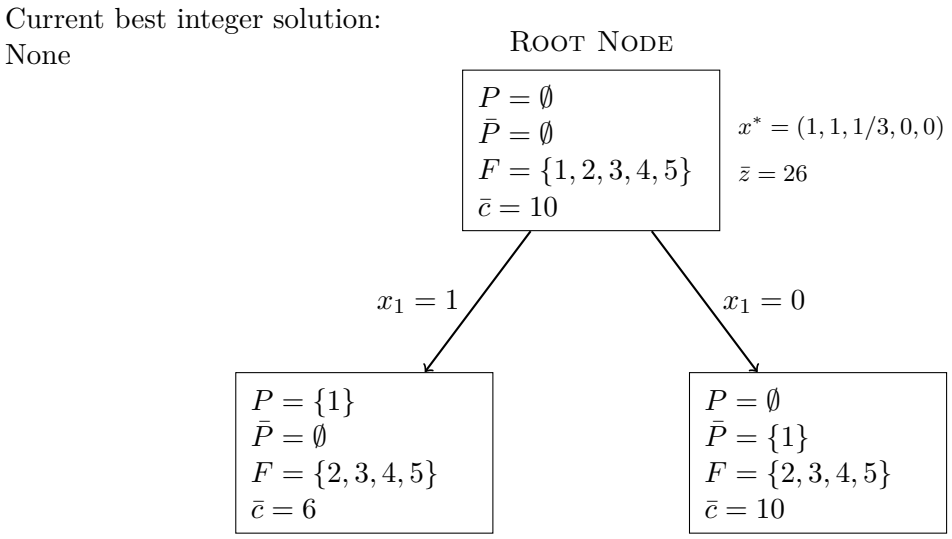

Figure 1: Branch-and-bound tree after exploring the root node.

We start from the root node, where all objects are free ($P = \bar{P} = \emptyset$, $F = J$). Remark that the objects are already sorted by decreasing density. We can quickly check that the critical object is $s = 3$ and the residual capacity is $c - w_1 - w_2 = 10 - 4 - 5 = 1$. Therefore, the optimal solution of the bounded continuous relaxation at the root node is

$$x_1^* = x_2^* = 1, \quad x_3^* = \frac{\bar{c}}{w_3} = \frac{1}{3}, \quad x_4^* = x_5^* = 0.$$

Its objective value is $12 + 12 + \frac{1}{3} \cdot 7 = 26.\bar{3}$ and, therefore, the upper bound is 26. Following the rule, we branch on x_1 as shown in Figure 1.

Next, we explore the node associated with $x_1 = 1$. Because $x_1^* = 1$ in the bounded continuous relaxation optimal solution at the root node, this node inherits the optimum from the father. Therefore, we branch on the next variable, x_2 and explore the node $x_2 = 1$. Again, $x_2^* = 1$ in the previous optimal solution to the bounded continuous relaxation; therefore, this node also inherits its father's optimal solution. We branch on x_3 and explore the node $x_3 = 1$. Remark that all three constraints $x_1 = 1$, $x_2 = 1$, and $x_3 = 1$ are active at this node. However, the total weight of the first three objects is $4 + 5 + 3 = 12$, which is larger than the capacity. Therefore, the problem is infeasible at the current node: we cannot simultaneously pack objects $1, 2, 3$. Consequently, all potential descendants of this node will be infeasible, and we shall not explore them. The BB tree at this point looks as in Figure 2.

No $x_j = 1$ node is open at this point, and we must explore the last $x_j = 0$ node we created. This is node $x_3 = 0$. The optimal solution of the bounded continuous relaxation is $x_1^* = x_2^* = 1$, $x_4^* = \frac{\bar{c}}{w_4} = \frac{1}{3}$. The corresponding objective value is $12 + 12 + \frac{1}{3} \cdot 6 = 26$. Once again we branch; this time, imposing $x_4 = 0$ or $x_4 = 1$. As you can imagine, because object 4 is the critical object at the current node, the child node with $x_4 = 1$ will be infeasible because there is insufficient capacity to pack objects $1, 2, 4$ simultaneously.

We shall then explore the node associated with $x_4 = 0$. The optimal solution of the bounded continuous relaxation is $x_1^* = x_2^* = 1$, $x_5^* = \frac{\bar{c}}{w_5} = \frac{1}{2}$. Its objective value is $12 + 12 + \frac{1}{2} \cdot 2 = 25$. Finally, we branch on variable x_5. Reasoning as before, we know that $x_5 = 1$ gives an infeasible node. At node $x_5 = 0$, we have fixed the value of all variables and, therefore, we obtain our first integer solution: $x_1^* = x_2^* = 1$, $x_3^* = x_4^* = x_5^* = 0$. Its objective value is $12 + 12 = 24$, and it

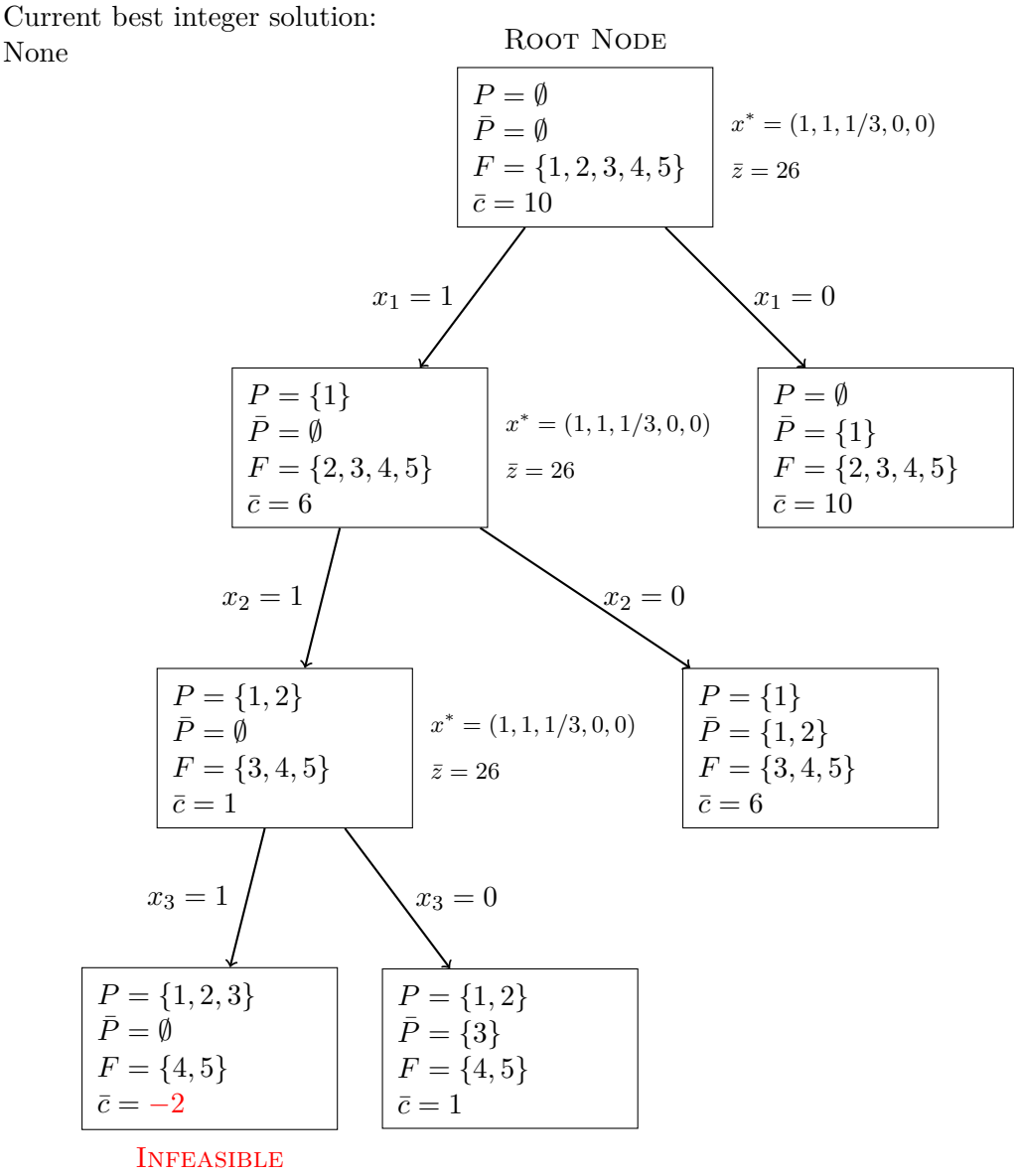

Figure 2: Branch-and-bound tree after encountering the first infeasible node.

constitutes the best (and only) lower bound available at the moment. The current state of the BB tree is depicted in Figure 3.

Two nodes remain open. We explore the latest one created, i.e., the one corresponding to $x_2 = 0$. The optimal solution of the bounded continuous relaxation is $x_1^* = x_3^* = x_4^* = 1$ and $x_2^* = x_5^* = 0$. The solution is integer; therefore, its objective value $z = 12 + 7 + 6 = 25$ is a lower bound for the optimal objective value of the overall problem. Moreover, this lower bound improves on the previous one (which was 24).

The current node is a leaf node, so we close it and move to the last open node, corresponding to $x_1 = 0$. The optimal solution of the bounded continuous relaxation is $x_2^* = x_3^* = 1$, $x_4^* = \frac{\bar{c}}{w_4} = \frac{2}{3}$, and $x_1^* = x_5^* = 0$. Its objective value is $\bar{z} = 12 + 7 + \frac{2}{3} \cdot 6 = 23$. This dual bound is worse than the current primal bound ($23 < 25$). Therefore, we can prune this node because no integer solution in the subtree rooted at the node can yield a better profit than our current best solution.

All nodes are now closed, and the algorithm terminates. The resulting final BB tree is shown in Figure 4.

2.5 Strengthening the dual bound

We now show how to derive a tighter dual (upper) bound, which also uses the optimal solution of the bounded continuous relaxation of the 01-KP. For simplicity of exposition, we consider the tree root node, i.e., the original 01-KP instance, without any branching constraint.

Let us focus on the critical object s. We don't know if s will be packed in an optimal solution of the integer 01-KP, but we can consider both possibilities (packed or not) and derive some conclusion in each of the two cases.

1. If s is not packed in the (as of yet unknown) optimal solution of the 01-KP, we could remove it from the instance, and nothing would change: we would eventually obtain the same optimal integer solution. However, removing s would alter the optimal solution of the bounded continuous relaxation of 01-KP. Missing s, in fact, $s + 1$ becomes the new critical object. Therefore, the dual bound from the bounded continuous relaxation would change as follows:

$$z^*_{\text{cont}} = \left\lfloor \sum_{j \in S, j < s} p_j + \frac{\bar{c}}{w_s} p_s \right\rfloor \quad \text{now becomes} \tag{5}$$

$$\bar{z}[x_s = 0] = \left\lfloor \sum_{j \in S, j < s} p_j + \frac{\bar{c}}{w_{s+1}} p_{s+1} \right\rfloor. \tag{6}$$

2. If s is packed in the optimal solution of the 01-KP (which we do not yet know), we can fix it and only consider solutions in which s is packed. Therefore, when solving the bounded continuous relaxation, we want to ensure that the entire object s is selected ($x_s = 1$). By definition of the critical object, however, not all objects $\{1, \ldots, s-1\}$ can also be packed because the capacity is insufficient. Therefore, objects $\{1, \ldots, s-1\}$ must "make room" for s and free up the $w_s - \bar{c}$ required capacity units (recall that $\bar{c} = c - \sum_{j \in J, j < s}$ is the residual capacity). Which object or objects should be removed to make room for s? The best we can do is to remove the worst possible object in terms of density, i.e., $s - 1$. If we remove $w_s - \bar{c}$ units of weight from the space occupied by $s - 1$, there is $w_{s-1} - (w_s - \bar{c})$ units of weight left for object $s-1$ and object s will fit in the knapsack. (We are assuming

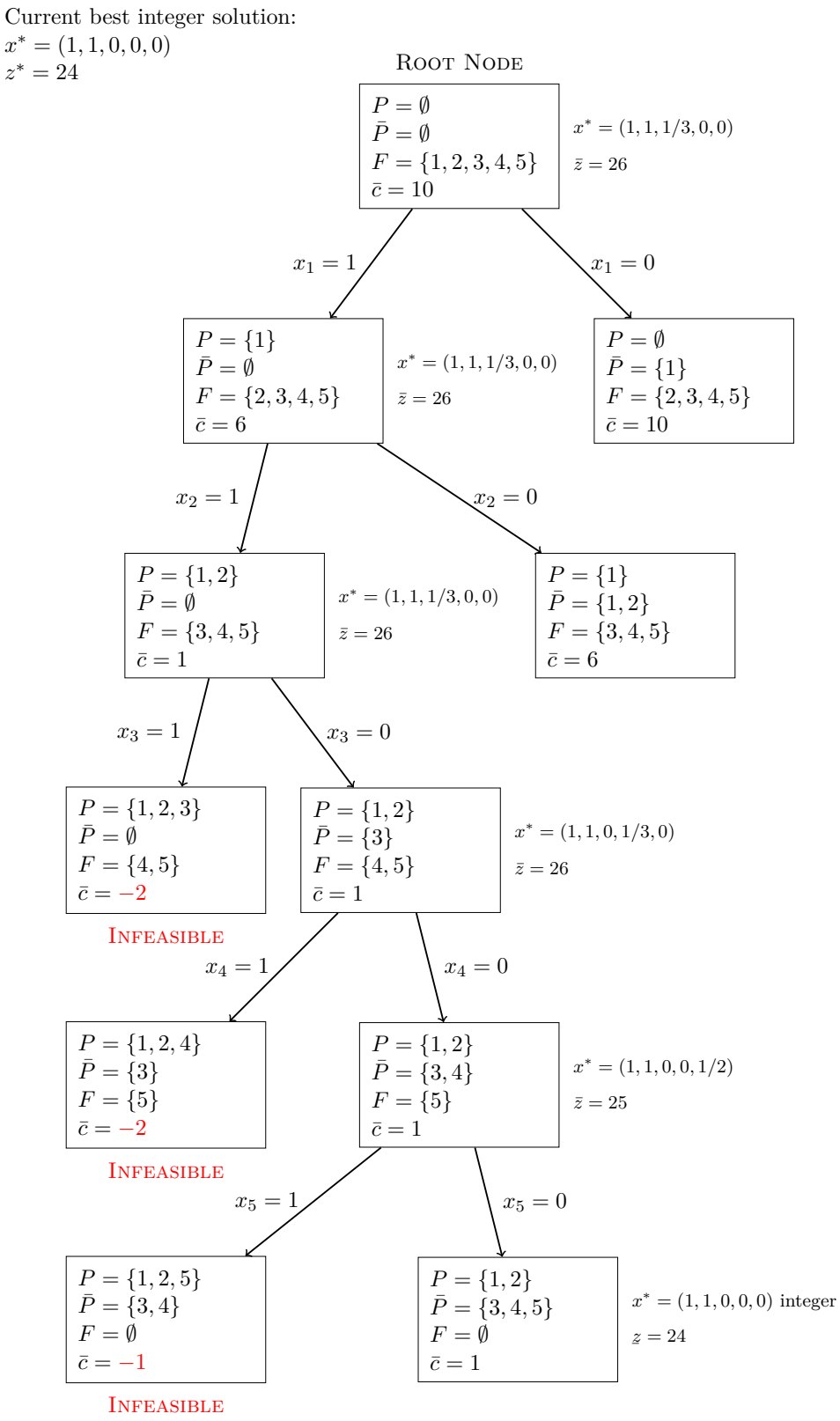

Figure 3: Branch-and-bound tree after encountering the first integer solution.

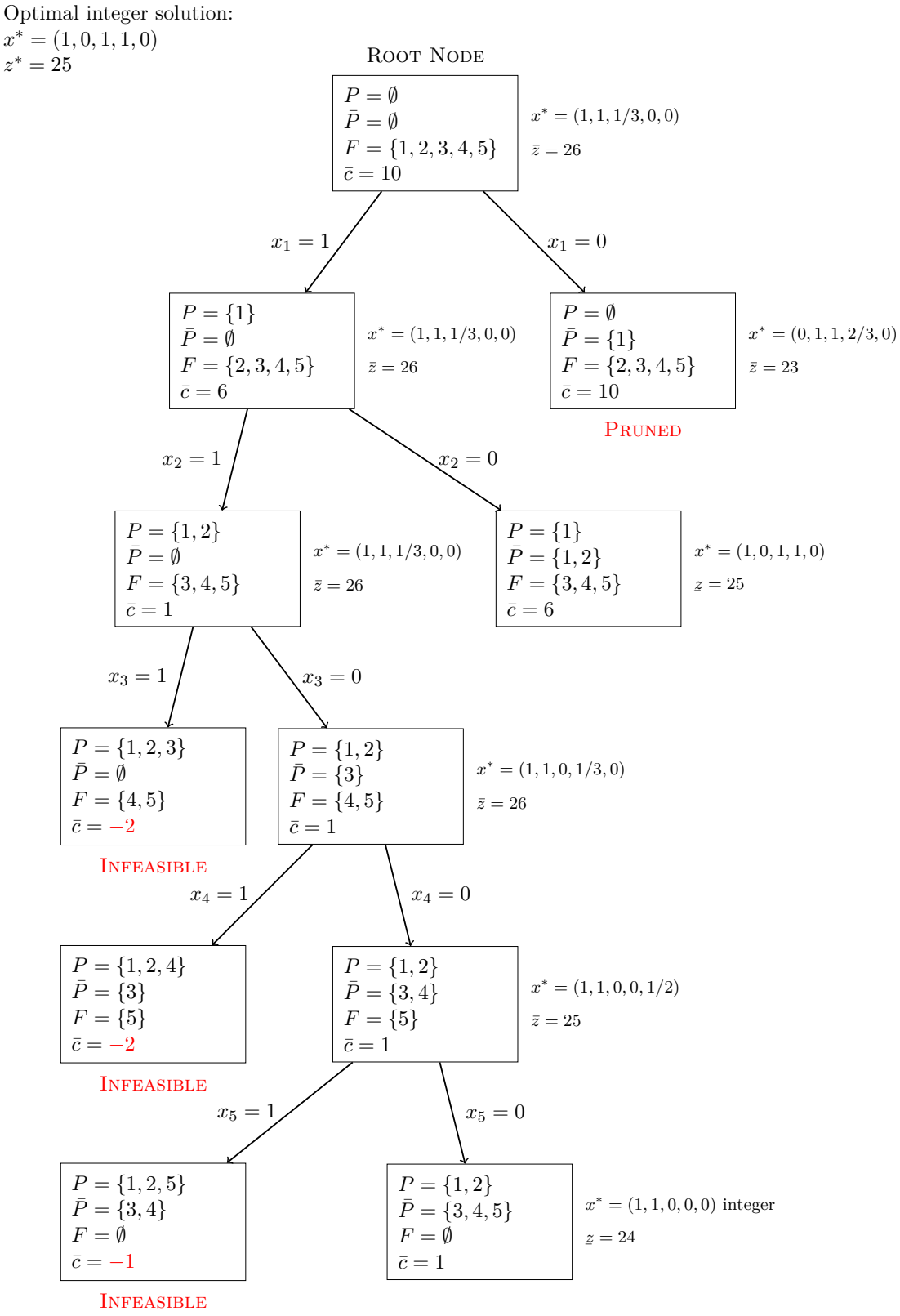

Figure 4: Branch-and-bound tree after completing the algorithm.

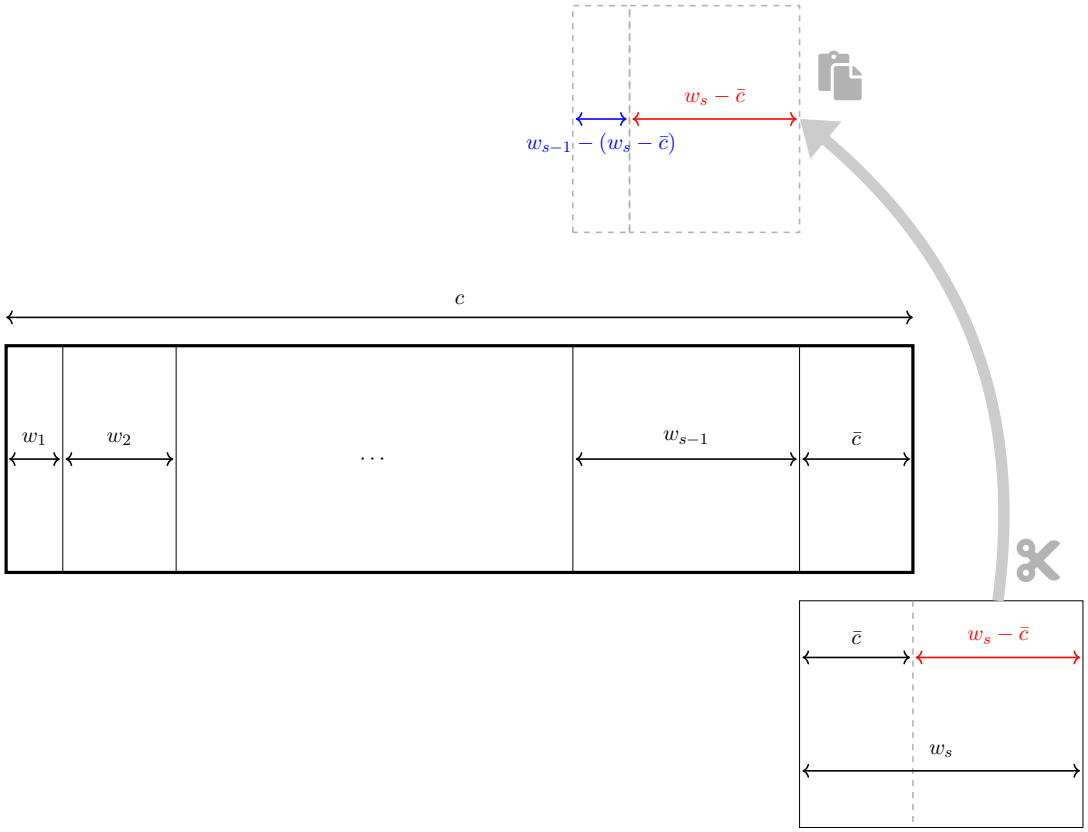

Figure 5: Upper bound strengthening when assuming that $x_s = 1$.

that the space w_{s-1} occupied by object $s-1$ is at least as large as the space $w_s - \bar{c}$ needed to accommodate object s; more on this later.) To ensure that object $s-1$ uses the assigned capacity units, we must impose

$$x_{s-1} w_{s-1} = w_{s-1} - (w_s - \bar{c}), \text{i.e., } x_{s-1} = 1 - \frac{w_s - \bar{c}}{w_{s-1}}.$$

Therefore, the dual bound from the bounded continuous relaxation would be:

$$\bar{z}[x_s = 1] = \left\lfloor \sum_{j \in S, j < s-1} p_j + \left(1 - \frac{w_s - \bar{c}}{w_{s-1}}\right) p_{s-1} + p_s \right\rfloor. \tag{7}$$

It remains to check that this bound is valid even when the required space $w_s - \bar{c}$ is larger than the space w_{s-1} occupied by object $s-1$. Intuitively, in this case, we should remove the entire object $s-1$ and, in part or in full, some of the objects $\{s-2, s-3, \dots, 1\}$ until we make enough space for object s. Although we do not do this, bound (7) is still a valid upper bound. The fraction of object $s-1$ which we must remove to make room for object s is $(w_s - \bar{c})/w_{s-1}$; let us denote this fraction with α. If $w_s - \bar{c}$ is larger than w_s, then $\alpha > 1$. This means that we are removing more than one copy of object $s-1$. For example, instead of removing the entire object $s-1$ and a fraction of object $s-2$, bound (7) corresponds to removing $\alpha > 1$ times object $s-1$. However, recall that the objects are sorted by density and object $s-1$'s density is not better than object $s-2$'s. Therefore, by removing $\alpha > 1$ times object $s-1$, we obtain a higher profit in (7) than by removing object $s-1$ and part of object $s-2$, which is consistent with devising a dual (upper) bound.

To summarise, denoting with z^{opt} the profit of the optimal solution of the integer 01-KP, we have derived the following bounds:

$$z^{\text{opt}} \leq \begin{cases} \bar{z}[x_s = 0] & \text{if object } s \text{ is not packed in the optimal solution of 01-KP,} \\ \bar{z}[x_s = 1] & \text{if object } s \text{ is packed in the optimal solution of 01-KP.} \end{cases}$$

However, a priori, we do not know whether object s will be part of the optimal solution of the integer 01-KP. Therefore, to obtain a valid upper bound on z^{opt}, we must be pessimistic and take the loosest of the two bounds, i.e.,

$$z^{\text{opt}} \leq \max\Big\{\bar{z}[x_s = 0],\ \bar{x}[x_s = 1]\Big\}. \tag{8}$$

We have replaced bound z^*_{cont} from the optimal solution of the bounded continuous relaxation with the more complex (8). In return, we should expect that this new bound is tighter. Indeed, the following theorem confirms this result.

Theorem 2. *Let* $\bar{z}_{\text{MT}} = \max\big\{\bar{z}[x_s = 0],\ \bar{x}[x_s = 1]\big\}$ *and* z^*_{cont} *be the objective value of the optimal solution of the bounded continuous relaxation. Then,* $\bar{z}_{\text{MT}} \leq z^*_{\text{cont}}$.

Proof. We will prove that $\bar{z}[x_s = 0] \leq z^*_{\text{cont}}$ and $\bar{x}[x_s = 1] \leq z^*_{\text{cont}}$. For the following, it will be useful to rewrite the bounds as follows:

$$z^*_{\text{cont}} = \sum_{j \in J, j < s} p_j + \left\lfloor \bar{c}\frac{p_s}{w_s} \right\rfloor \tag{9}$$

$$\bar{z}[x_s = 0] = \sum_{j \in J, j < s} p_j + \left\lfloor \bar{c}\frac{p_{s+1}}{w_{s+1}} \right\rfloor \tag{10}$$

$$\begin{aligned} \bar{z}[x_s = 1] &= \sum_{j \in J, j < s-1} p_j + \left\lfloor p_{s-1} - \frac{w_s - \bar{c}}{w_{s-1}} p_{s-1} + p_s \right\rfloor = \\ &= \sum_{j \in J, j < s-1} p_j + p_{s-1} + \left\lfloor p_s - (w_s - \bar{c})\frac{p_{s-1}}{w_{s-1}} \right\rfloor = \\ &= \sum_{j \in J, j < s} p_j + \left\lfloor p_s - (w_s - \bar{c})\frac{p_{s-1}}{w_{s-1}} \right\rfloor. \end{aligned} \tag{11}$$

In (9)–(11), we removed some integer terms from the floor function and rearranged some remaining terms. Recall that $p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$. Then, comparing (9) and (10), it follows immediately that $\bar{z}[x_s = 0] \leq z^*_{\text{cont}}$.

By definition of critical object, $w_s \geq \bar{c}$ and, again by the objects ordering, $\frac{p_{s-1}}{w_{s-1}} \geq \frac{p_s}{w_s}$. Therefore,

$$\begin{aligned} &(w_s - \bar{c})\left(\frac{p_{s-1}}{w_{s-1}} - \frac{p_s}{w_s}\right) \geq 0 && \Rightarrow \\ &w_s\frac{p_{s-1}}{w_{s-1}} - \cancel{w_s\frac{p_s}{w_s}} - \bar{c}\frac{p_{s-1}}{w_{s-1}} + \bar{c}\frac{p_s}{w_s} \geq 0 && \Rightarrow \\ &\bar{c}\frac{p_s}{w_s} \geq p_s - (w_s - \bar{c})\frac{p_{s-1}}{w_{s-1}}. \end{aligned} \tag{12}$$

Comparing (9), (11) and (12), it follows that $\bar{z}[x_s = 1] \leq z^*_{\text{cont}}$. □

Observe that the branch-and-bound algorithm works as long as we can compute a dual bound at each node. For general integer programmes, the dual bound is the objective value of the optimal solution of their continuous relaxation. However, the bound can also come from other sources. In our case, by exploiting the particular structure of the 01-KP, we can instead use the tighter bound $\bar{z}_{\text{MT}}$.

3 A dynamic programming algorithm for the 01-KP

In this section, we present an alternative algorithm to solve the 01-KP. To develop this algorithm, we use a technique called **Dynamic Programming** (DP), which is one of the fundamental tools in operational research. The main idea behind DP is that the optimal solution to a problem can be built using information from the optimal solutions of smaller problems. These solutions, in turn, are built using the optimal solutions to even smaller problems, etc. We can follow this pattern recursively until, eventually, we will have to solve a problem so small that its optimal solution is trivial. Then, retracing our steps will allow us to build the optimal solution to the original problem.

We want to apply this pattern to the 01-KP. The two characteristics that define the size of a 01-KP instance are the number of objects, n, and the knapsack capacity, c. Therefore, we will use optimal solutions for 01-KP instances with fewer elements and smaller knapsacks to finally build the optimal solution for the original instance.

Given two numbers, $m \in J$ and $q \in \{0, \dots, c\}$, let $z(m, q)$ denote the objective value of the optimal solution of the 01-KP "sub-instance" that only considers objects $\{1, \dots, m\}$ and whose knapsack has capacity q. Remark that the objective value of the optimal solution of the original instance can be written in this notation as $z(n, c)$. The central step in devising a DP algorithm is to write a formula for $z(m, q)$, which only depends on the problem parameters or on recursive calls to a fixed number of functions $z(m', q')$ where at least one of m' and q' is strictly smaller than, respectively m and q. In other words, we want to compute $z(m, q)$ using the function $z(\cdot, \cdot)$ applied to a smaller instance.

To devise such a formula for $z(\cdot, \cdot)$, let us reason on the last object of the corresponding sub-instance, i.e., object m. Reasoning in a way that reminds us of what we did in Section 2.5, we consider two cases:

1. If object m should be part of the optimal solution of the original 01-KP instance, we can simply fix it inside the knapsack and solve a residual sub-instance with objects $\{1, \dots, m-1\}$ and capacity $q - w_m$. In this case, we would obtain the profit p_m of object m, plus the best possible profit that can be obtained by an instance with objects $\{1, \dots, m-1\}$ and capacity $q - w_m$, i.e., $z(m-1, q-w_m)$.
2. If object m should not be part of the optimal solution of the original 01-KP instance, we can just toss it away and solve a sub-instance with objects $\{1, \dots, m-1\}$ and capacity q. In this case, we would not collect the profit associated with object m, but only the best possible profit that can be obtained by an instance with objects $\{1, \dots, m-1\}$ and capacity q, i.e., $z(m-1, q)$.

A priori, we do not know if object m should be part of the optimal solution of the 01-KP. Therefore, considering that our objective is to find the solution with the highest possible profit, we should try both options and choose the one yielding the best overall profit.

Formalising the above reasoning, we write the following **dynamic programming recursion**

(also known as Bellman equation):

$$z(m,q) = \max \begin{cases} z(m-1,q) & \text{if } w_m > q \\ \max\{\underbrace{p_m + z(m-1,q-w_m)}_{\text{Case 1.}}, \underbrace{z(m-1,q)}_{\text{Case 2.}}\} & \text{otherwise.} \end{cases} \tag{13}$$

The first line of (13) takes care of the case when object m is too large to fit into the currently considered knapsack (of capacity q) and, thus, cannot be packed. The second line chooses the value of $z(m,q)$ as the best among cases 1. and 2. discussed above. Note that we need the value of $z(m-1,q-w_m)$ and $z(m-1,q)$ to evaluate $z(m,q)$. If we do not know this value, we must recursively call function $z(\cdot,\cdot)$. When we successfully compute $z(m,q)$, we store this value in a table with n rows and c columns, where we can retrieve it if needed in future computations.

Because at least one of the two function parameters in the inner calls to $z(\cdot,\cdot)$ decreases, eventually, we will require the value of the function for parameters that are small enough to compute z trivially. The following **base conditions** can then be used:

$$z(1,q) = \begin{cases} p_1 & \text{if } w_1 \leq q \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

$$z(m,q) = 0 \quad \text{if } q < \min_{j=1,\dots,m} w_m. \tag{15}$$

Condition (14) states that, when attempting to pack object 1 only, we collect its profit (if the object fits in q capacity units) or do not get any profit (if the object does not fit). Condition (15) states that we cannot collect any profit if the capacity is so small that even the smallest object does not fit. Remark that this condition is also active when the capacity is negative, which might happen when evaluating $z(m-1,q-w_m)$ and $w_m > q$.

3.1 An example

To better understand how the DP algorithm works, we will solve a small example instance with $n = 4$, $c = 35$, and

$$(p_j)_{j\in J} = (8, 18, 20, 7)$$
$$(w_j)_{j\in J} = (7, 18, 22, 11).$$

The objective value of the optimal solution is given by $z(4,35)$. Using the DP recursion, we have:

$$z(4,35) = \max\{7 + z(3,24), z(3,35)\}.$$

Therefore, to compute $z(4,35)$, we must compute $z(3,24)$ and $z(3,35)$. Using the DP recursion again, we obtain:

$$z(3,24) = \max\{20 + z(2,2), z(2,24)\}$$
$$z(3,35) = \max\{20 + z(2,13), z(2,35)\}.$$

We continue to apply the recursion for all the unknown terms and use the base conditions.

$$\begin{aligned} z(2,2) &= 0 && \text{by (15)} \\ z(2,24) &= \max\{18 + z(1,6), z(1,24)\} \\ z(2,13) &= z(1,13) && \text{by (13) when } w_m > q \\ z(2,35) &= \max\{18 + z(1,17), z(1,35)\}. \end{aligned}$$

Finally,

$$z(1,6) = 0 \quad \text{by (14)}$$
$$z(1,24) = z(1,13) = z(1,17) = z(1,35) = 8 \quad \text{by (14)}.$$

We can proceed from bottom to top and replace calls to function $z(\cdot,\cdot)$ with the computed numeric values:

$$\begin{aligned}
z(2,24) &= \max\{18+0, 8\} = 18\\
z(2,12) &= 8\\
z(2,35) &= \max\{18+8, 8\} = 26\\
z(3,24) &= \max\{20+8, 18\} = 28\\
z(3,35) &= \max\{20+8, 26\} = 28\\
z(4,35) &= \max\{7+20, 28\} = 28.
\end{aligned}$$

The optimal solution for our instance has a profit of 28. To determine which objects are packed, we can trace back the recursive calls to $z(\cdot,\cdot)$. When the maximum of $z(m,q)$ is obtained by the part marked as "Case 1." in the second line of (13), object m is packed. When it is obtained via the first line or the part marked as "Case 2." in the second line, object m is not packed. Similarly, if $z(1,q) = p_1$ when evaluated as part of the calls that lead to the optimum, object 1 is packed; otherwise, it is not. In our case,

$$z(4,35) = \max\{27, \underbrace{28}_{\to z(3,35)}\} \qquad \text{(Object 4 not packed)} \tag{16}$$
$$z(3,35) = \max\{\underbrace{28}_{\to z(2,13)}, 26\} \qquad \text{(Object 3 packed)} \tag{17}$$
$$z(2,13) = z(1,13) \qquad \text{(Object 2 not packed)} \tag{18}$$
$$z(1,13) = 8 \qquad \text{(Object 1 packed).} \tag{19}$$

Therefore, the optimal solution packs objects 1 and 3.

4 Solutions

Exercise 1

Regarding point 1., we just observe that this problem can be formulated as a 01-KP. Formulation (1a)–(1c) is a valid formulation for the problem and does not need any variation.

Point 2. is more interesting. Remark that $x_i = 1$ if advertiser i's bid is accepted and $1 - x_j = 1$ if advertiser j's bid is not accepted. A naive attempt at modelling this problem, then, could be the following (here J denotes the advertisers set).

$$\max \quad \sum_{j \in J} p_j x_j + \sum_{(i,j) \in \mathcal{C}} \pi_{ij} x_i (1 - x_j) \tag{20a}$$
$$\text{subject to} \quad \sum_{j \in J} w_j x_j \leq c \tag{20b}$$
$$x_j \in \{0,1\} \qquad \forall j \in J. \tag{20c}$$

The problem with this formulation is that the objective function (20a) is not linear. Because we multiply variables, (20a) is quadratic. (The formulation would otherwise be correct.)

To obtain a linear formulation, we can proceed as follows. Let $z_{ij} \in \{0, 1\}$ be binary variables defined for each $(i, j) \in \mathcal{C}$. We want to attribute these variables the following meaning:

$$z_{ij} = \begin{cases} 1 & \text{if } i\text{'s bid is accepted but } j\text{'s is not} \\ 0 & \text{otherwise.} \end{cases} \tag{21}$$

If we can make these variables take the correct values, then we can rewrite the objective function as a linear function:

$$\max \quad \sum_{j \in J} p_j x_j + \sum_{(i,j) \in \mathcal{C}} \pi_{ij} z_{ij}.$$

Our mathematical programme is "blind", i.e., it does not know that we would like to interpret variables z_{ij} as described in (21). It is up to us to add the correct constraints that ensure that (21) holds. In particular, we want to *link* variables x and z such that (i) if $x_i = 1$ and $x_j = 0$, then $z_{ij} = 1$; (ii) in all other three cases ($x_i = x_j = 1$, $x_i = x_j = 0$, and $x_i = 0$ and $x_j = 1$), $z_{ij} = 0$. This aim can be achieved by adding the two following families of constraints to the model:

$$z_{ij} \leq x_i \qquad \forall (i, j) \in \mathcal{C} \tag{22}$$

$$z_{ij} \leq 1 - x_j \qquad \forall (i, j) \in \mathcal{C}. \tag{23}$$

Constraint (22) ensures that $z_{ij} = 1$ only if $x_i = 1$ and constraint (23) ensures that $z_{ij} = 1$ only if $x_j = 0$. Because both constraints hold simultaneously, z_{ij} can take value 1 only if both $x_i = 1$ and $x_j = 0$. Indeed, if $x_i = 0$, constraint (22) becomes $z_{ij} \leq 0$, i.e., since z_{ij} is binary, $z_{ij} = 0$. Analogously, if $x_j = 1$, constraint (23) becomes $z_{ij} \leq 0$, i.e., $z_{ij} = 0$.

The two constraints do not explicitly model the implication: "When $x_i = 1$ and $x_j = 0$, then $z_{ij} = 1$". The reason is that z_{ij} appears with a positive coefficient in the objective function. Therefore, this variable will take value 1 as soon as no constraint prevents it (because $z_{ij} = 1$ causes an increase of the maximised objective function). Consequently, we only need constraints that *prevent* z_{ij} from taking value 1 when it should not, but we do not need constraints that *force* z_{ij} to take value 1. Nonetheless, if we wanted to add such a constraint, we could do it: the formulation would still be correct, although it would include a superfluous constraint. As an exercise-in-the-exercise, you can verify that $z_{ij} \geq x_i - x_j \; \forall (i, j) \in \mathcal{C}$ is such a constraint.