ScuDo

Scuola di Dottorato ~ Doctoral School

WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation

Doctoral Program in Computer and Control Engineering (29^{th}cycle)

Knapsack Problems with Side Constraints

By

Rosario Scatamacchia

******

Supervisor(s):

Prof. Federico Della Croce di Dojola

Doctoral Examination Committee:

Prof. David Pisinger, Referee, Technical University of Denmark

Prof. Vincent T'Kindt, Referee, Ecole Polytechnique de l'Université de Tours

Prof. Arianna Alfieri, Politecnico di Torino

Prof. Andrea Grosso, Università di Torino

Prof. Ulrich Pferschy, Karl-Franzens University of Graz

Politecnico di Torino

2017

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and do not compromise in any way the rights of third parties, including those relating to the security of personal data.

Rosario Scatamacchia
2017

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Acknowledgements

This thesis is the result of my PhD in combinatorial optimization which I'd like to call a beautiful 3-year journey in the theory of computation. Among others, the theory of computation is about understanding the mathematical structures of optimization problems and how such structures can be exploited to solve these problems rather than to reveal their hardness. I found this context very fascinating. Moreover, it is a dynamic science producing fruitful collaborations among practitioners and researches from different disciplines such as computer scientists, physicists and mathematicians. I enjoyed having the opportunity of studying theories and approaches coming from different fields and backgrounds. For example, I did not expect at the beginning of my PhD that my favorite book in computer science would be a book written not by computer scientists but by physicists (namely «The Nature of Computation»[66], simply a beautiful book). I would like to mention what I loved most in my PhD by two quotes. The first one is about discovery:

> «A great discovery solves a great problem, but there is a grain of discovery in the solution of any problem. Your problem may be modest, but if it challenges your curiosity and brings into play your inventive faculties, and if you solve it by your own means, you may experience the tension and enjoy the triumph of discovery.» *George Pòlya*

I think this is definitely true. When you struggle with a problem, no matter if it is either a small or a big problem, you can really enjoy the pleasure of coming up with something new, something personal which was not out there before. The second quote is about the simplicity emerging from the hardness:

> «Simplicity is the final achievement. After one has played a vast quantity of notes and more notes, it is simplicity that emerges as the crowning reward of art.» *Frederic Chopin*

I think that Chopin's quote may be still valid in combinatorial optimization. When you deal with a problem having a very hard time to solve it, unexpected simple solutions may emerge along the way rewarding your work with the beauty of simplicity. Indeed, I experienced this in some of the problems presented in this thesis.

It is absolutely not easy to thank all the people I met within my PhD in few lines. First and foremost, I thank my tutor Prof. Federico Della Croce, who believed in me after an "out of the blue" e-mail that I sent to him when I was living in Toronto in 2013. I had left the academic world for 6 years at that time and he gave me the opportunity to follow my passion for applied mathematics. Federico taught me how to approach optimization problems. He guided me in these years enriching my PhD experience and his helpfulness and support go beyond the duties of a PhD tutor.
I am grateful to the SWARM group of the TIM Joint Open Lab of Turin which funded my PhD making all this possible. My colleagues from SWARM were supportive and helped me in all industrial projects I was involved in.
I want to thank Prof. Ulrich Pferschy from Karl-Franzens university of Graz. The 6–month research period I spent in Graz was a terrific experience. Ulrich has been a second supervisor to me with an uncommon generosity inside and outside the academic world.
A special thank goes to my colleagues of ALCO research group at DAUIN department: Michele Garraffa, Marco Ghirardi and Fabio Salassa. In different measure, I shared with them many moments and situations in the last three years and they always gave me their support.
I take the occasion to thank Roberto Aringhieri, Andrea Grosso and Pierre Hosteins, from the computer science department of Università di Torino. It has been rewarding and a pleasure to collaborate with them in my first works in combinatorial optimization.

I also thank all the nice guys I have known at Lab 8, at the other labs of DAUIN department and at TIM labs.

Finally, I dedicate this thesis to the three most important people of my life: to my loving girlfriend Elisa who unconditionally believes in me and with her love and truthfulness makes me feel loved as ever; to my mother Anna Maria and my brother Dario that are always on my side in all decisions I take.

Contents

List of Tables

Introduction

Everyday life aspects and directions are guided by decisions. Although passions and emotions play a fundamental role in taking small and big decisions, many contexts require a formalization of the decision-making process for solving complex problems. This is the goal of combinatorial optimization, which is a branch of operations research providing methods and tools for solving optimization problems arising in different fields. This thesis considers a specific class of optimization problems: the Knapsack Problems.

These are paradigmatic problems in combinatorial optimization where a set of items with given profits and weights is available. The aim is to select a subset of the items in order to maximize the total profit without exceeding a known knapsack capacity. The classical 0–1 Knapsack Problem (KP), in which each item can be picked at most once, was the first problem introduced in this class of optimization problems. Extensions of KP in different directions, such as modifications of the objective function or different constraints on the available item set, led to several relevant variants of practical interest. Knapsack Problems have been strongly investigated both from a theoretical and a practical point of view (we cite here, among others, two pioneering works [69]-[40], two books [47]-[61] and three comprehensive surveys [53], [58] and [81]). Although Knapsack Problems belong to the class of NP–hard problems and thus are intractable from a theoretical point of view [34], some of them are well-handled in practice. Very large instances can be solved in a reasonable time thanks to decades of research devoted to exploiting structures and properties of these problems. Knapsack Problems have straightforward applications in industrial contexts such as in cutting stock, scheduling cargo loading, project management, budget control, finance and in general in resource allocation problems. As an example, consider the problem of choosing a portfolio of investment projects

with expected profit/costs, given a monetary budget. The best combination of projects maximizing the profits corresponds to the solution of a knapsack problem where the items are the projects, the weights are their costs and the monetary budget is the knapsack capacity. Even everyday life situations relate to a knapsack problem like the change provided by an automatic coffee machine with the least number of coins available or the choice of the objects to carry for hiking excursion.
Besides, Knapsack Problems often arise as sub–problems in many more complex problems. Among others, KP emerges as a sub–problem in solving the Generalized Assignment Problem, which in turn is used for solving Vehicle Routing Problems [50]. Knapsack Problems also appear in graph partitioning problems [31], scheduling problems [39] and in cryptography [10]-[64].

The focus of the thesis is on four generalizations of KP involving side constraints beyond the capacity bound. More precisely, we provide solution approaches and insights for the following problems: The Knapsack Problem with Setups; the Collapsing Knapsack Problem; the Penalized Knapsack Problem; the Incremental Knapsack Problem. These problems reveal challenging research topics with many real–life applications.
The scientific contributions we provide are both from a theoretical and a practical perspective. On the one hand, we give insights into structural elements and properties of the problems and derive a series of approximation results for some of them. On the other hand, we offer valuable solution approaches for direct applications of practical interest or when the problems considered arise as sub–problems in broader contexts. As reported in the corresponding chapters, many contributions have been certified by publications in international journals and proceedings in conferences while others are about to be submitted to journals. The thesis is organized as follows:

In Chapter 1, we provide a survey of the research pursued for the classical 0–1 Knapsack Problem. Incidentally, some methods presented in this thesis rely also on ideas and methods originally designed for KP.

In Chapter 2, we consider the 0–1 Knapsack Problem with Setups (KPS). In this generalization of KP, items are grouped into families and if any item of a family is selected, this induces a setup cost as well as a setup resource

consumption. KPS has many applications of interest such as make–to–order production contexts, energy sector, cargo loading and product category management among others and more generally for resource allocation problems involving classes of elements. We introduce different solution approaches for this problem. More precisely, we propose an exact enumerative approach which handles the structure of the ILP formulation of KPS. It relies on partitioning the variables set into two levels and exploiting this partitioning. The proposed approach favorably compares to the algorithms in literature and to a commercial solver launched on the ILP formulation. It turns out to be very effective and capable of solving to optimality, within limited computational time, large instances with up to 100000 variables. We also introduce a new dynamic programming algorithm which performs much better than a previous dynamic program and turns out to be also a valid alternative to the exact enumerative approach. In addition, we provide further insights into KPS and derive a general inapproximability result. Furthermore, we investigate several relevant special cases which still permit fully polynomial time approximation schemes (FPTASs) and others where the problem remains hard to approximate.

In Chapter 3, we consider the 0–1 Collapsing Knapsack Problem (CKP) where the capacity of the constraint is not a scalar but a non–increasing function of the number of included items, namely, it is inversely related to the number of items placed inside the knapsack. Among others, CKP has wide applications such as in satellite communication and time–sharing computer systems, namely in problems where a structural overhead is induced by the number of items or users considered. On the one hand, we present a novel ILP formulation of CKP and an effective reduction procedure for restricting the solution space of the problem. The novel ILP constitutes a significant contribution for tackling the CKP since it makes possible to exploit the potentials of the modern IP solvers. On the other hand, we introduce an exact approach for CKP which is also extended to the multidimensional variants of CKP involving more than a capacity constraint (M-CKP). The approach relies on the ILP formulation of CKP and on an original branching scheme that induces the solution of several KPs (with the additional constraint that the number of items in the knapsack is fixed) by exploiting the particular structure of CKP. The proposed approach

favorably compares to the methods available in the literature and manages to solve to optimality very large size instances particularly for CKP and 2-CKP.

In Chapter 4, we deal with the 0–1 Penalized Knapsack Problem (PKP), where each item has a profit, a weight and a penalty. The problem calls for maximizing the sum of the profits minus the greatest penalty value of the items selected in a solution. PKP may have applications in resource allocation problems with a bi–objective function involving the maximization of the sum of the profits against the minimization of the maximum value of a feature of interest. PKP also arises as a pricing sub–problem in branch–and–price algorithms for the two–dimensional level strip packing problem in [54]. We provide theoretical results for the problem as well as an exact solution approach. More precisely, from the one hand we provide insights into the problem and a characterization of its linear relaxation. We also derive a surprising negative approximation result. On the other hand, we propose an exact approach relying on a procedure which narrows the relevant range of penalties and on dynamic programming algorithms. The proposed approach turns out to be very effective in solving hard instances of PKP and compares favorably both to commercial solver CPLEX 12.5 applied to the ILP formulation of the problem and to the best available exact algorithm in the literature.

In Chapter 5, we investigate the 0–1 Incremental Knapsack Problem (IKP). In this generalization of KP the capacity grows over time periods. If an item is placed in the knapsack in a certain period, it cannot be removed afterwards. The problem calls for maximizing the sum of the profits over the whole time horizon. IKP has many real-life applications since, from a practical perspective, it is often required in allocation resource problems to deal with changes in the input conditions and/or in a multi–period optimization framework.
We propose a series of results extending the contributions currently available in the literature. In particular, we manage to prove the tightness of some approximation ratios of an approximation algorithm presented in [37]. Then, we devise a Polynomial Time Approximation Scheme (PTAS) when the input value associated with the time periods is considered as a constant. Also, we consider a restricted variant of the problem with two time periods and where

each item can be packed in the first period. Under this assumption, we derive an algorithm with a constant approximation factor of $\frac{6}{7}$.

In Chapter 6, we provide the conclusions of the thesis and sketch possible future developments.

A part of the contents presented in chapters 2, 4, 5 was developed within two research periods (January-March and September-December 2016) spent at Karl-Franzens University of Graz in Austria, in the Statistics and Operations Research Department led by Prof. Ulrich Pferschy.

The work presented in the thesis was supported by a fellowship from TIM Joint Open Lab SWARM (Turin, Italy). SWARM Lab studies and develops solutions for problems involving group dynamics arising in complex systems. In particular, the focus is on innovation projects concerning the application of distributed and pervasive technologies using the paradigm of cooperation and collaboration, where "things" or "people" with devices interact among them within complex environments. The activities I carried out in the SWARM Joint Open Lab ranged in different contexts of practical relevance. The specific interest to the Knapsack Problems came from connections with industrial research projects within the smart-home and smart-city paradigms.

The work on the 0–1 Knapsack Problem with Setups has been partially supported by FLEXMETER, FLEXible smart METERing for multiple energy vectors with active prosumers, funded by the European Commission under H2020, Grant Agreement N. 646568 (see [1]). FLEXMETER project involves the efficient management of the buildings energy consumptions. Within the project thematics, a problem concerning the proper management of energy peak demands originated a practical application of KPS, as discussed in Chapter 2.

Also a smart city project within the Internet of Things paradigm shared connections with knapsack problems. The project involved the study of solutions for the so-called opportunistic Internet of Things. The Internet of Things expression refers to the network of physical objects and devices provided with sensing and computing capabilities. A challenging task could be the collection of data from these objects since it is not always possible to have a specific network infrastructure for connecting them. An emerging trend in the IoT

is the design of solutions based on ad hoc connections provided by mobile devices (e.g. smartphones) by opportunistically considering daily routines or habits of the users. I contributed to investigate the problem of covering garbage bins (equipped with sensors of their filling level) by exploiting the passage of pedestrians or vehicles. In this respect, a link with Knapsack Problems is to see users as knapsacks with a given capacity (time window) which can carry items (bins to cover) with different weights (covering times of the bins). An introductory work on the matter is presented in [20].

The 0–1 Knapsack Problem

1.1 Introduction

In this first chapter, we provide an overview of the fundamental properties of the 0–1 Knapsack Problem as well as of the solution approaches proposed in the literature. The overview is based on the valuable survey in [81] and the comprehensive book [47]. A particular emphasis is paid to concepts and methods which have been exploited in the following of this thesis.

1.2 Basic concepts

1.2.1 Integer Linear Programming formulation

The 0–1 Knapsack Problem can be formally stated as follows: a capacity value c and a set of n items j with weight w_j and profit p_j are given. The objective is to choose a subset of the items to maximize the profits such that the total weight of the items does not exceed c. The problem can be formulated as the following Integer Linear Programming (ILP) model (denoted (KP))

(KP):

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j \tag{1.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c \tag{1.2}$$

$$x_j \in \{0,1\} \qquad j = 1,\ldots,n \tag{1.3}$$

where binary variables $x_j = 1$ iff item j is placed in the knapsack. In order to avoid meaningless cases, the following assumption are made: $\sum_{j=1}^{n} w_j > c$; $w_j \leq c \quad j = 1,\ldots,n$.
Indeed, instances of KP violating the first condition have the trivial solution in which all items are picked. Any item violating the second condition can be trivially removed from the problem. Without loss of generality (W.l.o.g.), it is also assumed that w_j, p_j and c are positive integers. Fractional values can be scaled to integer values by multiplication by suitable factors, while instances with non positive coefficients can be handled as outlined in [35]. We will denote the maximum profit and the maximum weight of the items by p_{max} and w_{max} respectively.

For the sake of readability, throughout the thesis, for each considered problem we will denote the optimal solution value by z^* and the optimal solution vector by x^*. Likewise, we will use the general notation z^{LP} and x^{LP} to denote the optimal solution value and vector of the Linear Programming relaxations. Some symbols or letters may recur in the different chapters with a different meaning. For the mathematical formulations of the problems, we will usually stick to the notation used in the literature.

1.2.2 Reference instances

We briefly sketch the main classes of instances of KP. The generation schemes of these instances are commonly employed in the construction of challenging instances for other knapsack problems as well. In a nutshell, different classes of instances involve a different correlation between the profits and the weights of the items. Profits and weights range in the interval $[1, R]$, with R positive integer in general equal to 1000 or 10000. The capacity value is usually proportional to the sum the weights of the items by a factor < 1. The main groups of KP instances are:

1. *Uncorrelated instances*: p_j and w_j are randomly distributed in $[1, R]$. These instances reflect situations where profits are reasonably independent from the weights. They are expected to be easy to solve since the presence of very appealing items (high profit, low weight) and particularly worse items (low profit, high weight).

2. *Weakly correlated instances*: weights w_j randomly range in $[1, R]$ while the profits p_j range in $[w_j - \frac{R}{10}, w_j + \frac{R}{10}]$ with $p_j \geq 1$. These instances illustrate situations in project management and in finance where the expected return of an investment is proportional to its cost with some variations.

3. *Strongly correlated instances*: These instances involve a greater level of correlation with weights w_j randomly distributed $[1, R]$ and $p_j = w_j + \frac{R}{10}$. In real-life applications, the profit of a project may be equal to the investment plus an additional charge.

4. *Inverse strongly correlated instances*: profits p_j range in $[1, R]$ and $w_j = p_j + \frac{R}{10}$. These instances are similar to the strongly correlated ones but the fixed contribution is negative.

5. *Almost strongly correlated instances*: Profits are strongly correlated to weights with some additional noise. Weights w_j range in $[1, R]$ and p_j in $[w_j + \frac{R}{10} - \frac{R}{500}, w_j + \frac{R}{10} + \frac{R}{500}]$.

6. *Subset sum instances*: weights w_j range in $[1, R]$ and $p_j = w_j$. For these instances, the goal is to fill the knapsack as much as possible.

7. *Uncorrelated instances with similar weights*: weights w_j are distributed in a narrow interval with large extremes $[100000, 100100]$ while profits p_j range in $[1, 1000]$.

1.2.3 Linear relaxation

In the linear relaxation of KP, denoted as KP^{LP}, the integrality constraints (1.3) are replaced by the inclusion in the interval $[0, 1]$. KP^{LP} has a special

structure and can be solved in the following straightforward way. First, the items are sorted according to non–increasing ratios of profits over weights:

$$\frac{p_1}{w_1} \geq \frac{p_2}{w_2} \geq \cdots \geq \frac{p_n}{w_n} \tag{1.4}$$

The ratio $\frac{p_j}{w_j}$ represents the profit per weight unit of each item j and it is also called *efficiency* of item j. The ordering (1.4) will be assumed throughout the chapter if not stated otherwise. According to this greedy order of efficiency, items $j = 1, 2, \ldots$ are then inserted into the knapsack as long as

$$\sum_{k=1}^{j} w_k \leq c. \tag{1.5}$$

Let us denote as the *split item* the first item s which cannot be packed. This item is also called *break item* or *critical item*. The optimal solution vector x^{LP} is given by setting $x_j^{LP} = 1$ for $j = 1, \ldots, s-1$, $x_j^{LP} = 0$ for $j = s+1, \ldots, n$ and $x_s^{LP} = (c - \sum_{j=1}^{s-1} w_j)/w_s$.
The optimal solution value z^{LP} is:

$$z^{LP} = \sum_{j=1}^{s-1} p_j + (c - \sum_{j=1}^{s-1} w_j)\frac{p_s}{w_s} \tag{1.6}$$

This procedure is also known as Dantzig's rule [21]. A graphical proof of correctness is outlined in [21]. For a formal proof see [61]. Computing the optimal solution is x^{LP} is easy and requires $O(n)$ after the ordering of the items (1.4), which requires $O(n \log n)$. Nevertheless, authors in [6] showed how to solve KP^{LP} in $O(n)$ without any sorting of the items by an advanced partitioning procedure.

1.2.4 Quick–and–dirty heuristics

We outline here the main heuristic algorithms proposed for KP. Exact solution approaches are discussed more in detail in Sections 1.3–1.5. Henceforth, we

indicate the value of any feasible solution of KP as a lower bound for the problem.

A simple heuristic solution is given by setting to zero the value of the break item in the optimal solution of KP^{LP}. This solution is known as the *split solution* (or *break solution*) and has a profit and weight equal to $\hat{p} = \sum_{j=1}^{s-1} p_j$ and $\hat{w} = \sum_{j=1}^{s-1} w_j$ respectively. An improved heuristic consists in starting from the *split solution* and in adding the items $j = s+1, \ldots, n$ one after the other as soon as they fit in the residual capacity. This heuristic is known as the *Greedy* algorithm and has a solution value greater than or equal to $\hat{p}$. The quality of the solution provided by the *Greedy* algorithm with respect to the optimal solution is discussed in Section 1.6. The running time of this heuristic is dominated by the sorting of the items, thus its complexity is $O(n \log n)$.

We mention two other straightforward heuristics introduced in [75]. These heuristics are particularly suitable for strongly correlated instances. The *forward greedy algorithm* adds a single item to the split solution taking the best objective value:

$$z^f = \max_{j=s+1,\ldots,n} \{\hat{p} + p_j \mid \hat{w} + w_j \leq c\} \tag{1.7}$$

The *backward greedy algorithm* adds the split item to the break solution and removes one item retaining the best objective value:

$$z^b = \max_{j=1,\ldots,s-1} \{\hat{p} + p_s - p_j \mid \hat{w} + w_s - w_j \leq c\} \tag{1.8}$$

For both the heuristics, finding the maximum values requires $O(n)$.

1.2.5 Upper Bounds

Considerable research has been devoted to compute upper bounds on KP as close to the optimal solution as possible. These upper bounds are exploited in algorithmic frameworks such as branch and bound and dynamic programming. A natural upper bound is given by the optimal solution of the continuous

relaxation z^{LP}. Given the integrality of the profits, a valid bound is given by rounding down z^{LP}:

$$U_1 = \left\lfloor z^{LP} \right\rfloor \tag{1.9}$$

Another bound can be derived from the Lagrangian Relaxation of KP, which corresponds to solve the following problem

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j + \lambda(c - \sum_{j=1}^{n} w_j x_j) \tag{1.10}$$

$$\text{subject to} \quad x_j \in \{0,1\} \quad \forall j = 1,\ldots,n \tag{1.11}$$

where $\lambda \geq 0$ is the non negative multiplier associated with the capacity constraint. Nevertheless, solving the Lagrangian Relaxation yields a bound greater than or equal to z^{LP} (see, e.g., [87], Theorem 10.3). A solution value equal to z^{LP} is attained by setting $\lambda = \frac{p_s}{w_s}$.

In [59], the following upper bound U_2 based on packing the split item or not is derived:

$$U_2 = \max\left\{\left\lfloor \hat{p} + (c-\hat{w})\frac{p_{s+1}}{w_{s+1}} \right\rfloor, \left\lfloor \hat{p} + p_s + (c-\hat{w}-w_s)\frac{p_{s-1}}{w_{s-1}} \right\rfloor\right\} \tag{1.12}$$

The first term in (1.12) considers the case in which $x_s = 0$. The residual knapsack capacity $(c-\hat{w})$ is filled with the most efficient item x_{s+1} not packed in the break solution. Here, the inclusion of x_{s+1} in $\{0,1\}$ is replaced by $x_{s+1} \geq 0$. The second term refers to the case where $x_s = 1$. By definition, the split item added to the split solution exceeds the knapsack capacity $(c-\hat{w}-w_s < 0)$. A valid bound for KP is obtained by removing the most effective item before x_s without any restriction, namely $x_{s-1} \geq 0$, until the capacity excess is 0. Clearly, in any optimal solution the break item x_s is either 0 or 1. Therefore, taking the maximum between the two terms in (1.12) yields a valid upper bound for KP. We note that $U_2 \leq U_1$. Running the algorithm in [6], U_2 can be computed

in $O(n)$ since the algorithm identifies also items $s+1$ and $s-1$ together with the split item s.

Extending this principle, an upper bound based on a partial enumeration procedure is proposed in [60]. Let us denote by M a subset of the items $j = 1,\ldots,n$. Also, let us denote by $X_M = \{x_j \in \{0,1\}, j \in M\}$ the set of the 0–1 vectors representing the values of the variables x_j, with $j \in M$. Since every optimal solution of KP must be originated in one of the vectors in M, an upper bound on KP is given by

$$U_M = \max_{\tilde{x} \in X_M} \{U(\tilde{x})\} \tag{1.13}$$

where $U(\tilde{x})$ is a generic upper bound for KP with $x_j = \tilde{x_j}$ and $j \in M$. The complexity of calculating U_M is equal to the complexity of computing $U(\tilde{x})$ times $O(2^{|M|})$, thus this bound may make sense only for relatively small sets M. Generally speaking, tighter values of the bound are obtained when M consists of items with efficiency similar to that of the break item. Other bounds following a similar reasoning were laid out in [29], [28] and [67].

Finally, we mention further approaches for deriving bounds for KP which rely on adding valid inequalities to KP^{LP}. These inequalities reduce the solution space of KP^{LP} without discarding optimal solutions of model (KP). We cite the studies of the facets of the knapsack polytope in [5], [36], [86] and the bounds derived in [5] and [62] by working out cardinality constraints on the number of items.

1.2.6 Variable Reduction

Methods and techniques have been proposed for reducing the size of KP instances as well. The first reduction algorithm was presented in [43]. This and other similar reduction methods are based on computing upper bounds when a variable x_j is set either to 0 or 1. If one of the upper bounds of this dichotomic search is less than a lower bound available, the variable x_j can be fixed at its optimal value. Let us denote by LB a lower bound for KP found by some heuristic and by u_j^0 an upper bound for KP without item j (namely for

model (KP) with the additional constraint $x_j = 0$). Similarly, let us denote by u_j^1 an upper bound for KP when item j is packed (model (KP) with the constraint $x_j = 1$). The set N^1 of the variables x_j which can be fixed to 1 is

$$N^1 = \{j = 1, \ldots, n \mid u_j^0 < LB + 1\}. \tag{1.14}$$

In fact, if discarding an item would not lead to a solution better than the incumbent solution, item j must be necessarily part of a possible improving solution. The term LB + 1 comes from the integrality of the input data. Analogously, the set N^0 of the variables x_j which can be fixed to 0 is

$$N^0 = \{j = 1, \ldots, n \mid u_j^1 < LB + 1\}. \tag{1.15}$$

Fixing the variables in N^0 and N^1 yields a reduced KP instance hopefully easier to solve than the original one. Given this general picture, the time complexity of a reduction scheme depends on the computational effort required to compute the upper bounds. In [27], the following bounds are derived for each item j

$$u_j^0 = \hat{p} - p_j + (c - \hat{w} + w_j)\frac{p_s}{w_s} \quad j = 1, \ldots, s-1 \tag{1.16}$$

and

$$u_j^1 = \hat{p} + p_j + (c - \hat{w} - w_j)\frac{p_s}{w_s} \quad j = s+1, \ldots, n \tag{1.17}$$

where the integrality constraint on the split item is replaced with $x_s \geq 0$. Notice that these upper bounds are based on the optimal solution of the linear relaxation. Therefore, computing u_j^1 for $j < s$ and u_j^0 for $j > s$ is meaningless since additional constraints on the variables do not affect z^{LP}.
Once the split item is known, each bound can be calculated in constant time. Therefore, the complexity of the overall reduction is $O(n)$. The reduction scheme proposed in [43] provides tighter bounds but with a higher complexity $O(n^2)$. An improved reduction approach running in $O(n \log n)$ is devised in [60].

1.2.7 Core Problem

Computational experiments carried out over many different test instances indicated that usually only a relatively small subset of the items is critical for finding an optimal solution of KP. More precisely, the empirical evidence suggests that items with high efficiency $\frac{p_j}{w_j}$ are very likely to be included in the knapsack while items having a low efficiency are not expected to be part of an optimal solution. Basically, the crucial decision may concern the selection of items with efficiency close to that of the split item $\frac{p_s}{w_s}$. These items constitute the so-called *core* of a knapsack problem. The first definition of the core was presented in [6]. This definition assumes the knowledge of the optimal solution vector x^* and the sorting of the items by efficiency. Let us define indexes j_1 and j_2 as

$$j_1 = \min\{j : x_j^* = 0)\}; \tag{1.18}$$

$$j_2 = \max\{j : x_j^* = 1)\}. \tag{1.19}$$

The core is constituted by the items in the interval $C = \{j_1, \ldots, j_2\}$ and the corresponding core problem is formulated as

$$\text{maximize} \quad \sum_{j \in C} p_j x_j + P \tag{1.20}$$

$$\text{subject to} \quad \sum_{j \in C} w_j x_j \leq c - W \tag{1.21}$$

$$x_j \in \{0,1\} \quad \forall j \in C \tag{1.22}$$

where $P = \sum_{j=1}^{j_1-1} p_j$ and $W = \sum_{j=1}^{j_1-1} w_j$.

For different classes of instances (with the exception of the strongly correlated instances) the size of the core is expected to be a relatively small fraction of the number of the items. Therefore, with an a–priori knowledge of j_1 and j_2 we could reasonably solve the core problem by algorithms like branch and

bound or dynamic programming. Clearly, we don't have this a–priori knowledge. The core can be only "guessed" by algorithms and, due to this uncertainty, we also need to validate the setting of the variables outside the core. Nonetheless, exploiting the core concept constitutes a fundamental ingredient for the most effective exact solution approaches of the 0–1 Knapsack Problem, as discussed in Section 1.5.

The main advantage of focusing on a core problem is that high quality (or optimal) solutions could be reached quickly. On one hand, having good lower bounds in the early stages may enhance the performances of algorithms in solving the core problem with a limited computational effort. On the other hand, high quality solutions make reduction rules more effective in fixing the variables outside the core. In addition, since only a subset of the items is considered, the sorting of all items can be avoided. This is beneficial for solving instances, such as the uncorrelated instances, where the computational time required for sorting all items dominates the overall solution time.

Nevertheless, a degeneration issue may arise in algorithms relying on the core concept, as pointed out in [77]. Items in the core have similar efficiency levels, so basically solving the core problem amounts to getting a filled knapsack. When the core is composed of items with similar (or proportional) weights, it may be difficult to reach a good solution which fills the knapsack capacity (for further details on the matter, see [77]). In these cases, algorithms suffering from the absence of a good lower bound, like a branch and bound algorithm, could get stuck in the complete enumeration of the variables in the core with computational time $O(2^{|C|})$.

In the next sections, we provide a general picture of the main algorithms and methods designed for the exact solution of KP: Dynamic programming, Branch and Bound and Core based algorithms.

1.3 Dynamic programming algorithms

Many combinatorial optimization problems are difficult to solve because their parts interact. A decision made in one part of the problem influences the

other parts with unpredictable consequences. However, in some circumstances it is possible to identify a "channel" through which these interactions are limited. In particular, a global optimal solution can be reached by defining suitable sub–problems and combining their optimal solutions. This is the principle of dynamic programming, which is a technique applied in many areas of combinatorial optimization. The seminal book by Bellman [7] is the most valuable reference for an introduction in this field.

1.3.1 Dynamic programming by weights and by profits

In the classical dynamic program designed for KP, called also *dynamic programming by weights*, the idea is to consider sub–problems induced by a subset of the items and by a capacity value less or equal than c. The dynamic program works under the assumption that coefficients w_j and c are integer. Suppose that, for a subset of the items, we are given the optimal solutions of the corresponding sub–problems for all capacities values from 1 up to c. Then, an item is added and the solutions for this enlarged subset are easily computed by considering the previous solutions. The procedure iterates until all items are considered thus delivering a global optimal solution.

More precisely, let us denote by $z_j(d)$ the optimal solution value of the knapsack sub–problem consisting of items $1,\dots,j$ and capacity $d \le c$. Once $z_{j-1}(d)$ has been calculated for all capacities $d = 1,\dots,c$, the solution values $z_j(d)$ are computed according to the following recursion

$$z_j(d) = \begin{cases} z_{j-1}(d) & \text{if } d < w_j, \\ \max\{z_{j-1}(d), z_{j-1}(d-w_j)+p_j\} & \text{if } d \ge w_j. \end{cases} \tag{1.23}$$

for $j = 1,\dots,n$ and $d = 1,\dots,c$. The recursion formula (1.23) is known as *Bellman Recursion* and has a straightforward meaning. If $d < w_j$, this means that item j cannot be part of a sub–problem with capacity less than its weight, therefore the optimal solution for the sub–problem does not change. Conversely, if item j fits in the knapsack, there are two possibilities: either item j is discarded or it is picked. In the former case, clearly the solution $z_j(d)$ is equal

to $z_{j-1}(d)$. In the latter, item j contributes to the objective function with its profit p_j decreasing the capacity available to $d-w_j$. By definition, the best solution value for the sub–problem with capacity $d-w_j$ and items $1,\ldots,j-1$ is given by $z_{j-1}(d-w_j)$. Therefore, taking the maximum between $z_{j-1}(d)$ and $z_{j-1}(d-w_j)+p_j$ yields the optimal solution value for $z_j(d)$.

After an initialization step where $z_0(d)=0$ for $d=1,\ldots,c$, recursion (1.23) is applied for $j=1,\ldots,n$. Eventually, the optimal solution value is given by $z_n(c)$. It is easy to see that, for each item, we compute the optimal solution values of c sub–problems in constant time. Thus, the overall complexity of the algorithm is $O(nc)$ establishing a pseudopolynomial algorithm for KP. In a naive implementation of the algorithm, the complexity in space is also $O(nc)$ since we have to store all the solution values in the iterations.

The optimal solution set can be obtained by a simple backtracking strategy. It is sufficient to introduce 0–1 pointers $A_j(d)$ encoding the information whether item j has been placed into the knapsack sub–problem with capacity d ($A_j(d)=1$) or not ($A_j(d)=0$), namely

$$A_j(d)=\begin{cases}1 & \text{if } z_j(d)=z_{j-1}(d-w_j)+p_j,\\ 0 & \text{if } z_j(d)=z_{j-1}(d).\end{cases} \tag{1.24}$$

The optimal solution is reconstructed in the following way. If $A_n(c)=1$, item n is part of the optimal solution and then we check $A_{n-1}(c-w_n)$. If $A_n(c)=0$, item n is not in the optimal solution, thus we analyze $A_{n-1}(c)$. Following this reasoning, the overall solution set is computed. In this implementation, a further table to store the values of the pointers is needed.

We mention a general procedure proposed in [70] for reducing the space requirements and computing the optimal solution set in dynamic programming algorithms without increasing the overall running time. This generalized scheme is based on a recursive divide and conquer strategy which breaks down a problem in smaller and smaller sub–problems until it becomes easy to identify the optimal solution sets of the sub–problems at hand. These sets are then combined to reach a global solution. Among the different applications of

interest, the space complexity of the dynamic programming for KP is reduced to $O(n+c)$.

The dynamic programming by weights has been the first approach proposed for the exact solution of the knapsack problem. There exists also a *dynamic programming by profits* which is also useful for deriving approximation schemes as described in Section 1.6. Since the algorithm involves similar arguments as for dynamic programming by weights, we just outline the necessary definitions and formulas.
The main idea is to define sub–problems with a subset of items where a profit level must be reached with the minimum weight. Assuming integer profits, let us denote by $y_j(q)$ the minimum weight of the subset of items $1,\dots,j$ to reach a total profit q, with $q=1,\dots,U$ and U indicating an upper bound on the objective function. If no solution exist for $y_j(q)$, we set $y_j(q)=c+1$. First, we initialize $y_0(0)=0$ and $y_0(q)=c+1$ for $q=1,\dots,U$. Then, the following recursion is iteratively applied for $j=1,\dots,n$ and $q=1,\dots,U$

$$y_j(q)=\begin{cases} y_{j-1}(q) & \text{if } q<p_j, \\ \min\{y_{j-1}(q), y_{j-1}(q-p_j)+w_j\} & \text{if } q\geq p_j. \end{cases} \tag{1.25}$$

The optimal solution value corresponds to $\max\{y_n(q) \mid y_n(q)\leq c\}$ and the time complexity of the dynamic programming by profit is $O(nU)$.

1.3.2 Dynamic programming with states

We outline here a technique which does not improve the worst case complexity $O(nc)$ of the dynamic programming by weights but often performs much better in practice. The idea is to see each pair $(d,z_j(d))$ as a state $(\overline{w}_j,\overline{p}_j)$ where $\overline{p}_j$ denotes the profit reachable with the first j items and capacity $\overline{w}_j$. For each item j, we can introduced a list of states

$$L_j=[(\overline{w}_{1j},\overline{p}_{1j}),(\overline{w}_{2j},\overline{p}_{2j}),\dots,(\overline{w}_{mj},\overline{p}_{mj})] \tag{1.26}$$

with the number of states bounded by c. The states in a list may be reduced applying the following *dominance rule*:

Given two generic states $(\overline{w},\overline{p})$ and $(\overline{w'},\overline{p'})$, if $\overline{w} < \overline{w'}$ and $\overline{p} \geq \overline{p'}$ or $\overline{w} \leq \overline{w'}$ and $\overline{p} > \overline{p'}$, the state $(\overline{w'},\overline{p'})$ is said to be dominated by the state $(\overline{w},\overline{p})$ and it will never be part of an optimal solution.

Many states may be hopefully discarded according to this principle. Therefore, if we replace the array representation of all capacity values with a list representation, we may deal with only a limited number of states during the iterations with consequent improvements of the running time.

The corresponding algorithm is denoted as *Dynamic programming with states* (or with lists). In the initialization step, we have a list L_0 only with the state (0,0). Then, we progressively add items as in the previous dynamic programming algorithms. Given a list L_j, we can straightforwardly derive the list L_{j+1} for item $j+1$. First, a list L'_{j+1} is created by a componentwise addition of the pair (w_{j+1},p_{j+1}) with L_j:

$$L'_{j+1} = L_j \oplus (w_{j+1},p_{j+1}) \tag{1.27}$$

Then, the two list are merged by applying the dominance rule so as to obtain L_{j+1}. Without going into details, we point out that all these operations can be performed with complexity $O(c)$ by keeping the states ordered by nondecreasing weights. Hence, iterating through all items yields an overall complexity $O(nc)$. We observe that the total number of states is also bounded by $O(2^n)$. We reasonably assume that the complexity $O(nc)$ is lower than $O(2^n)$.

From a practical perspective, it is also useful to combine the dominance concept with upper bounds to further reduce the number of states. An upper bound for a state $(\overline{w}_j,\overline{p}_j)$ of item j can be derived by considering the sub–problem with items $j+1,\ldots,n$, initial profit $\overline{p}_j$ and capacity $c-\overline{w}_j$. If the upper bound computed is lower than an incumbent solution available, the state is eliminated.

In the light of these considerations, the dynamic programming with states can be expected to deliver practical performances much more effective than the worst case complexity $O(nc)$.

1.3.3 Primal–Dual dynamic programming

The *Bellman recursion* computes an optimal solution for KP from scratch by iteratively adding items. As pointed out in Section 1.2.7, generally only few items around the split item have a different value in the optimal solutions of KP and KP^{LP}. This motivates the *primal–dual dynamic programming* introduced in [78]. The expression *primal–dual* refers to the fact that also infeasible solutions are accepted in the recursions while the previous approaches consider only feasible solutions and they are also called *primal* methods.

Let us denote by $z_{a,b}(d)$ the optimal solution of the problem:

$$\text{maximize} \quad \sum_{j=1}^{a-1} p_j + \sum_{j=a}^{b} p_j x_j \tag{1.28}$$

$$\text{subject to} \quad \sum_{j=a}^{b} w_j x_j \leq d - \sum_{j=1}^{a-1} w_j \tag{1.29}$$

$$x_j \in \{0,1\} \quad \forall j = a,\ldots,b \tag{1.30}$$

In practice, $z_{a,b}(d)$ is the optimal solution value of the knapsack problem where items $j < a$ are placed into the knapsack while items $j > b$ are discarded. The recursion formulas of the *primal–dual dynamic programming* are

$$z_{a,b}(d) = \begin{cases} z_{a,b-1}(d) & \text{if } d - w_b < 0, \\ \max\{z_{a,b-1}(d), z_{a,b-1}(d-w_b) + p_b\} & \text{if } d - w_b \geq 0, \end{cases} \tag{1.31}$$

$$z_{a,b}(d) = \begin{cases} z_{a+1,b}(d) & \text{if } d + w_a > 2c, \\ \max\{z_{a+1,b}(d), z_{a+1,b}(d+w_a) - p_a\} & \text{if } d + w_a \leq 2c. \end{cases} \tag{1.32}$$

with $a = 1,\ldots,s$; $b = s-1,\ldots,n$ and $d = 0,\ldots,2c$. Recursion (1.31) refers to the possible selection of item b in the knapsack while recursion (1.32) evaluates the removal of item a from the knapsack.

The initialization step sets $z_{s,s-1}(d) = -\infty$ for $d = 0,\dots,\hat{w}-1$ and $z_{s,s-1}(d) = \hat{p}$ for $d = \hat{w},\dots,2c$, where $\hat{w}$ and $\hat{p}$ are the weight and the profit of the split solution introduced in Section 1.2.4. After that, the recursion (1.31) and (1.32) are alternatively used in order to compute $z_{s,s}(\cdot), z_{s-1,s}(\cdot), z_{s-1,s+1}(\cdot),\dots$ until $z_{1,n}(\cdot)$ is calculated. The optimal solution value corresponds to $z_{1,n}(c)$.

In other words, the enumeration starts from the split item and takes in account the split solution. Then, it proceeds either by adding an item $b \geq s$ in the knapsack or by removing an item $a < s$. The complexity of the algorithm is again $O(nc)$ as for the dynamic programming by weights. However, applying the recursions with states, dominance rule and upper bounds, the practical running time can decrease considerably. Empirical evidence showed that in general only a small number of items around the split item has to be considered. This concepts have been also applied within the *Minknap* algorithm proposed in [76], which is one of the most effective algorithm for KP (see Section 1.5.2).

1.3.4 Balancing, Horowitz and Sahni Decomposition

Finally, we cite other two techniques outlined for dynamic programming algorithms. The first one is *balancing*. Roughly speaking, a balanced solution is a solution with a weight sufficiently close to the capacity bound, namely its weight is not lower than the capacity by more than the weight of a single item. Indeed, any optimal solution is balanced and so it may be appealing to consider balanced states only. A balanced dynamic program is proposed in [78]. The algorithm is capable of solving many knapsack problems in linear time if profits and weights of the items are bounded by a constant. For KP, the dynamic program runs with complexity $O(np_{max}w_{max})$. Consequently, balanced dynamic programming may be attractive whenever the size of profits and weights of the items is reasonable limited.
Another method proposed in [40] is based on the decomposition of the initial KP with n items in two sub–problems of equal size $\frac{n}{2}$. Each sub–problem is solved by dynamic programming with states and the results are easily combined to find an optimal solution for the original KP. The number of states in each sub–problem is bounded by $O(2^{\frac{n}{2}})$. Hence, the running time is given by the minimum between $O(nc)$ and $O(2^{\frac{n}{2}})$, which constitutes an improvement by a

square root over the complete enumeration $O(2^n)$.
We mention the recent research line devoted to the design of exact exponential algorithms for NP-hard problems. This field has been developed since the last decade and involves the construction of exact algorithms with an exponential complexity but which improve on the brute–force search. Incidentally, among the main techniques, the *sort–and–search* paradigm ([52]) revisits the idea in [40] of decomposing a problem in two sub–problems. The performance comparison of the algorithms is according to the $O^*(\cdot)$ notation where, roughly speaking, the polynomial contributions to the final complexity are disregarding. The focus is in fact on the exponential contribution which gives the asymptotic behavior of the worst case performance for large n. Although the progresses made in the field for many combinatorial optimization problems, for KP no improvement of the complexity bound $O(2^{\frac{n}{2}})$ has been obtained so far. For a general introduction on exact exponential algorithms, see [85, 32].

1.4 Branch and Bound algorithms

The first branch and bound algorithm for KP is proposed in [49] and many approaches developed afterwards resemble this framework. The most successful variants iteratively consider the most efficient items as candidate for branching. In particular, a branch and bound algorithm is proposed in [40]. This algorithm is denoted as the Primal–Branch algorithm in [47]. Its recursive formulation is sketched in Algorithm 1. Each iteration corresponds to a branch operation on the most effective free item x_f. The node with $x_f = 1$ is considered first than the node with $x_f = 0$. Let us denote by p' and w' the profit and the weight in a node given by the previous branching on variables x_j $(j < f)$. The algorithm backtracks either if $w' > c$ (line 2 of Algorithm 1) or if no additional item can fit in the residual capacity, namely $c - w' < \underline{w}_f$, with $\underline{w}_f = \min_{j=f,\dots,n} w_j$ (line 4 of Algorithm 1) . The algorithm may also backtrack if an upper bound computed for the node is less than a current lower bound LB (lines 5–6 of Algorithm 1) .

Algorithm 1 Primal–Branch(f,p',w'):boolean

1: $improved$ = false;
2: **if** $w' > c$ **then return** $improved$; **end if**
3: **if** $p' > LB$ **then** $LB = p'$; $f^* = f$; $improved$ = true; **end if**
4: **if** $f > n$ or $c - w' < \underline{w}_f$ **then return** $improved$; **end if**
5: Compute an upper bound U' with capacity $c - w'$;
6: **if** $p' + U' \leq LB$ **then return** $improved$; **end if**
7: **if** Primal–Branch($f+1$,$p'+p_f$,$w'+w_f$) **then** $x_f = 1$; $improved$ = true; **end if**
8: **if** Primal–Branch($f+1$,p',w') **then** $x_f = 0$; $improved$ = true; **end if**
9: **return** $improved$;

Before launching Primal–Branch(1,0,0), LB and variables x_j are set to 0. Notice that the optimal solution vector x^* can be defined in constant time during the backtracking operations through the boolean variable *improved*. In fact, when an improved solution is found (line 3 of Algorithm 1) in a node, Primal–Branch algorithm then returns *true* in all the parents of the node inducing the updating of corresponding variables x_j (lines 7–8 of Algorithm 1). Tracking the index f^* when the optimal solution is obtained, the optimal solution set is simply given by setting $x^*_j = x_j$ for $j = 1, \ldots, f^* - 1$ and $x^*_j = 0$ for $j = f^*, \ldots, n$.

The ideas of the "lazy" updating of the solution vector and of backtracking when $c - w' < \underline{w}_f$ are introduced in [59]. Authors in [59] propose a branch and bound algorithm, the *MT1* algorithm, which is based on the same framework of the Primal–Branch algorithm but relies also on a new dominance step among items. Upper bound U_2 is used in the iterations of the algorithm. In [61], a comparison among different branch and bound schemes showed that in general the *MT1* algorithm runs with the lowest running times.

Eventually, we mention the *primal–dual branch* algorithm introduced in [75]. In a nutshell, the rationale of the algorithm is related to the idea of balancing cited in Section 1.3.4. The *primal–dual branch* algorithm considers knapsack solutions "appropriately filled" by inserting or removing items around the split item and the split solution. Contrary to the previous branch and bound algorithms, the *primal–dual* algorithm also goes over infeasible solutions.

1.5 Core based algorithms

The most successful algorithms for KP are based on the core concept introduced in Section 1.2.7. As said, the definition of the core is based on the knowledge of the optimal solution. The core is only guessed by algorithms usually through the choice of 2δ delta items around the split item. The initial step of the algorithms is the search of an approximated core.
The first methods for finding a core were presented in [6] and [60]. In [75], a modification of the classical *quicksort* algorithm ([38]) effectively determines a core $C = \{s-\delta, \ldots, s+\delta\}$ around the split item s of size $2\delta+1$. The algorithm, denoted as *Find–Core*, recursively partitions intervals of items considering their efficiency levels. The algorithm resembles the *quicksort* algorithm except for the fact that only the intervals including the split item are partitioned further. This characteristic reduces the complexity of the algorithm to $O(n)$. The discarded intervals constitute a partial ordering of the items by efficiency and this information may be also used in algorithmic frameworks. Items in any interval are not sorted but there is an ordering among the intervals. The final outcome of the *Find–Core* procedure is the core C and intervals of items saved in two stacks H and L. Items in an interval H have a higher efficiency than the one of the items in the core while items in L have a lower efficiency. The ordering of the intervals by non–increasing efficiency of the items may be represented as follows:

$$\overbrace{H_1, H_2, \ldots, H_H}^{\text{H}}, C, \overbrace{L_l, \ldots, L_2, L_1}^{\text{L}} \tag{1.33}$$

Some of the main algorithms based on the core concept are discussed in the following.

1.5.1 *MT2* algorithm

The *MT2* algorithm is introduced in [60]. The algorithm is based on computing and exploiting a fixed core. Its main steps can be summarized as follows:

1. An approximated core $C = \{s-\delta, \ldots, s+\delta\}$ is first computed. The core size $\mid C \mid$ is equal to n for $n < 100$, otherwise we set $\mid C \mid = \sqrt{n}$.

2. Items in the core are ordered by efficiency and the core problem is solved by *MT1* algorithm. Let denote by z_C^* the corresponding optimal solution value.

3. The enumerative upper bound U_C is calculated (as U_M in Section 1.2.5). If $U_C \leq \sum_{j=1}^{s-\delta-1} p_j + z_C^*$ the core solution is optimal and the algorithm terminates.

4. The reduction scheme in [60] fixes the variables outside the core. If not all variables can be fixed, the remaining variables are sorted by efficiency and the induced sub–problem is solved by *MT1* algorithm.

Other algorithms with a fixed core are presented in [6] and [29]. A description of all these algorithms and some computational tests are provided in [61]. A special variant of *MT2* algorithm, called *MTh*, is proposed in [62]. For this algorithm, the use of cardinality constraints and other techniques may allow a drastic improvement the computational performance of *MT2*.

1.5.2 *Minknap* algorithm

Since the core is hard to estimate, algorithms based on a variable expanding core have been proposed in [75] and in [76]. We provide here a more detailed description of the *Minknap* algorithm in [76] which constitutes also the backbone of the current state-of-the-art *Combo* algorithm. The main idea is to compute a core with only the split item through the *Find–Core* procedure and to iteratively add items considering the intervals in stacks H and L. The solution space of the expanding core is explored by a dynamic programming with states. An effective procedure for analyzing dominance among states combined with the use of suitable upper bounds makes the algorithm very effective. Moreover, reduction and sorting operations are performed only when they are needed. We sketch the main features of the algorithm in the following:

1. First, the *Find–Core* algorithm finds the core $C = \{s\}$ as well as the stacks of intervals H and L.

2. At each iteration, two items h and l are considered for expanding the core to the left and to the right respectively. These items belong to the current set of items sorted by efficiency outside the core: h is the item in H with the smallest efficiency while l is the item in L with the largest efficiency. The corresponding variables are equal to the value in the split solution, namely $x_h = 1$ and $x_l = 0$. A dynamic programming with states alternatively evaluates the insertion of item l in the knapsack or the removal of item h. Each state is denoted as a triple representing the profit, the weight and a partial representation of the corresponding solution set of packed items. The third element is used to efficiently recover the optimal solution set. Basically, the dynamic program implements the recursions (1.31) and (1.32) and adds items h and l to the core. If there is no such item h or l in the set of sorted variables, the next interval respectively from H or L is selected. The reduction scheme in [27] is applied to fix variables in the interval. If there are variables not fixed, these are ordered and added to the set of sorted items.

3. Before the inclusion in the recursions of item h or l, an enumerative upper bound is computed considering all undominated states. A reduction test is then performed in order to avoid the inclusion of the items in the core. Computing this upper bound is time–consuming since we have to go through all states of the dynamic program. At the same time, it may limit the size of the core considerably thus yielding savings in the overall running time.

4. States within dynamic programming can be eliminated by the dominance criterion or by an upper bound test. Upper bounds on states are computed in constant time by considering the most effective items on the right or on left of the core according to whether the weight of the state is less or greater than the capacity value.

5. The algorithm terminates when all states have been eliminated or all items outside the core have been fixed. The optimal solution vector is

reconstructed by an efficient procedure based on consecutively solving a limited number of knapsack problems.

The *Minknap* algorithm is proven to yield the smallest symmetrical core around the split item with respect to a fixed–core algorithm ([76]).
Finally, we cite the *Expknap* algorithm provided in [75]. The algorithm is very similar to *Minknap* and relies on the use of the *primal–dual branch* algorithm instead of dynamic programming. *Minknap* can be seen as a breadth–first search variant of *Expknap* and this feature is crucial for properly handling the structure of the expanding core.

1.5.3 *Combo* algorithm

The current state-of-the-art algorithm for KP is presented in [57]. The algorithm is called *Combo* since it combines different techniques and methods previously developed and it is very effective in solving large KP instances. We just mention the features of the algorithm since essentially it is based on the structure of *Minknap*. The dynamic programming introduced in *Minknap* starts with an initial core with up to eight items. This core is built with some heuristic rules. The core is then expanded and additional techniques are further introduced if the problem seems to become difficult to solve. These techniques rely on the greatest common divisor of the weights of the items, on the introduction of minimum and maximum cardinality constraints and on pairing the dynamic programming states with items not included in the core.
The criterion used as measure of the problem hardness is the number of states in the dynamic program. The comparison with three threshold values establishes which technique should be employed during the iterations of the algorithm.

In [57], an extensive performance analysis of the most effective exact algorithms for KP is presented. More precisely, algorithms *Minknap*, *MTh* and *Combo* were tested over several classes of instances with up to 10000 items. The results showed a comparable performance of *Minknap* and *MTh* which solved many instances in limited computational time (less than one second). However, there are instances where the algorithms ran into difficulties (e.g. in strongly correlated instances for *Minknap* and in even-odd instances for *MTh*). *Combo*

exhibited a stable performance and strongly outperformed both *Minknap* and *MTh*, successfully solving all instances in few milliseconds at most. For further details on the matter, see [57].

A final remark is that, although the algorithms proposed in the literature as well as nowadays ILP solvers manage to effectively solve large KP instances, the research for the knapsack problem cannot be considered concluded. In [79], computational tests with the same algorithms showed that KP is still hard to solve for many different instances. These instances were built either by considering standard instances with profits and weights distributed in larger intervals, or by designing new classes of instances which compromise the effectiveness of the commonly employed techniques for computing upper bounds on KP.

1.6 Approximation schemes

For various reasons, such as limited resources or time, one might be interested in suboptimal solutions of a problem rather than getting an optimal solution at all costs. Clearly, it is desirable that the objective value of these solutions is relatively close to the optimal solution value and to be able to measure the gap of the computed solution from the optimum. This is the general principle of approximation algorithms. For the knapsack problem, a lot of research has been devoted to this topic. In this section, we outline the features of the main approximation algorithms devised for KP. In addition, we discuss general conditions to derive approximation schemes according to the results presented in [83].

1.6.1 Approximation algorithms for KP

We first recall the basic definition of an approximation algorithm with a relative performance guarantee for a maximization problem. Consider an algorithm A computing a solution value z^A for a problem with optimal solution value z^*.

Definition 1. *Algorithm A is an approximation algorithm with relative performance guarantee r (< 1) if*

$$\frac{z^A}{z^*} \geq r \tag{1.34}$$

holds for all problem instances.

Thus, algorithm A always provides solutions whose value is at least a given percentage of the optimal value. The relative performance guarantee r is also called approximation ratio.

The *Greedy* algorithm introduced in Section 1.2.4 provides a solution resembling the optimal solution of the linear relaxation. One might think that the solution value obtained by *Greedy* is reasonably not far from the optimum for any instance of KP. Unfortunately, this is not the case.

Consider the following instance with $n = 2$, $c = M$, $p_1 = 2$, $w_1 = 1$, $p_2 = w_2 = M$ and M being an arbitrary positive integer number. According to the efficiencies of the items, *Greedy* will pack item 1 only obtaining a solution with value 2. The optimal solution is given by selecting item 2 and has value M. Therefore, for large values of M, the performance of the heuristic is arbitrarily bad.

However, these pathological situations can be avoided by taking the maximum between the profits attained by *Greedy* and the maximum profit of the items p_{max}. Denote the corresponding value by z^G. From the results of the linear relaxation of KP in Section 1.2.3, we have:

$$z^* \leq z^{LP} \leq \sum_{j=1}^{s-1} p_j + p_{max} \leq z^G + z^G = 2z^G \tag{1.35}$$

Hence, this slight modification of *Greedy* algorithm reaches always a profit equal to at least one half of the optimal value and therefore constitutes an algorithm with a relative performance guarantee of $\frac{1}{2}$. It is easy to show that this performance guarantee is also tight. Consider a slight modification of the previous instance with entries: $n = 3$, $c = 2M$, $p_1 = 2$, $w_1 = 1$, $p_2 = w_2 = p_3 = w_3 = M$. The algorithm will select item 1 and either item 2 or item 3 getting a solution with value $2 + M$. The optimal solution consists in packing items 2 and 3 and has value $2M$, which implies that the approximation ratio goes arbitrarily close to $\frac{1}{2}$ as M increases.

Recall now the definition of an ε–approximation scheme where the relative performance guarantee is not constant but depends on a parameter ε.

Definition 2. *Algorithm A is an ε–approximation scheme if, for every $\varepsilon \in (0,1)$,*

$$z^A \geq (1-\varepsilon)z^* \tag{1.36}$$

holds for all problem instances.

An ε–approximation scheme is a *Polynomial Time Approximation Scheme* (PTAS) if its running time is polynomial in n, whereas it is a *Fully Polynomial Time Approximation Scheme* (FPTAS) if the running time is polynomial both in n and $\frac{1}{\varepsilon}$.

All PTASs for KP follow the idea of "guessing" a certain number of items with the largest profits included in the optimal solution by going through feasible tuples of items. The residual capacity in the sub–problems is filled in a heuristic way. All algorithms require a space complexity $O(n)$. We mention the classical PTAS proposed in [84] which runs with complexity $O(n^{\frac{1}{\varepsilon}})$. An improved PTAS is presented in [12], which has an approximation ratio $\frac{k+1}{k+2}$ and complexity $O(n^k)$ for some parameter k.

FPTASs for KP are derived by applying a scaling technique on the profits of the items. We illustrate the basic method to construct an FPTAS. First, we scale the profits p_j and consider new profits $\tilde{p}_j = \left\lfloor \frac{p_j}{K} \right\rfloor$ with a constant K to be appropriately chosen. Then, we simply run the dynamic programming by profits on the problem with scaled profits. Notice that any feasible solution for this problem is also feasible for KP. Denote by $\tilde{X}$ the optimal solution set for the scaled instances and by z^A the corresponding solution value in the original problem, i.e. $z^A = \sum\limits_{j \in \tilde{X}} p_j$. The optimal solution set for KP is denoted by X^*. Then, the following series of inequalities holds:

$$z^A = \sum_{j \in \tilde{X}} p_j \geq \sum_{j \in \tilde{X}} K \left\lfloor \frac{p_j}{K} \right\rfloor \geq \sum_{j \in X^*} K \left\lfloor \frac{p_j}{K} \right\rfloor \geq \sum_{j \in X^*} K \left(\frac{p_j}{K} - 1 \right) = z^* - K\,|X^*| \tag{1.37}$$

In particular, the second crucial inequality is due to the fact $\tilde{X}$ is an optimal solution for the problem with scaled profits, whereas X^* is not necessarily optimal for this problem. In order to obtain the performance guarantee of $(1-\varepsilon)$, we must have:

$$K \leq \frac{\varepsilon z^*}{|X^*|} \tag{1.38}$$

Since the cardinality of the optimal solution set X^* is bounded by n and $p_{max} \leq z^*$, condition (1.38) is easily satisfied by setting $K = \varepsilon \frac{p_{max}}{n}$.
The running time of the dynamic programming is $O(n\tilde{U})$, where $\tilde{U}$ is a generic upper bound on the scaled problem. A simple upper bound is given by $n\tilde{p}_{max} \leq n\frac{p_{max}}{K} = \frac{n^2}{\varepsilon}$. The overall running time is $O(n^3\frac{1}{\varepsilon})$ thus establishing an FPTAS for KP. The space complexity is also $O(n^3\frac{1}{\varepsilon})$.
Improving on this basic approach, FPTASs for the knapsack problem were given in [41], [51], [55]. The current best performing FPTAS is provided in [45, 46] with complexity $O(n\min\{\log n, \log(\frac{1}{\varepsilon})\} + \frac{1}{\varepsilon^2}\log(\frac{1}{\varepsilon})\min\{n, \frac{1}{\varepsilon}\log(\frac{1}{\varepsilon})\})$ in time and $O(n+\frac{1}{\varepsilon^2})$ in space.

1.6.2 About the existence of an FPTAS and inapproximability

While KP admits a basic FPTAS, in general it may be not immediate to derive an FPTAS even for apparently slight variations of KP. In some cases, it is also unclear if any approximation algorithm could exist at all. We briefly describe straightforward conditions provided in [83] for establishing the existence of an FPTAS for a general family of problems called *subset selection problems*. We report a definition of these problems from [73], which is a variant of the general definition given in [83].

Definition 3. *([73]) A subset selection problem is defined by a ground set X with n elements each of which has associated a positive profit $p(x)$ for $x \in X$ and for each subset Y of X it can be decided in polynomial time whether Y is feasible. Moreover assume that every instance of the subset selection problem has a feasible solution. Then we are looking for a feasible subset of X with maximum total profit.*

The following theorem about the existence of an FPTAS based on a pseudopolynomial algorithm holds:

Theorem 1. *([83]) If there exists an exact algorithm for a subset selection problem with running time polynomial in n and in $\sum_{x \in X} p(x)$ then there exists also an FPTAS for this problem.*

It easy to see that KP is a subset selection problem and it is solvable by a pseudopolynomial algorithm with a polynomial running time in the number of variables and profits (i.e. the dynamic programming by profits). At the same time, if a KP variant (or another problem) does not belong to this class, this may be taken as a clue, but not a proof, that an FPTAS and/or approximation algorithms might not exist.
Negative approximation results are also worthy since they provide insights into the structural difficulty of a problem, as we shall see in Chapter 2 and Chapter 4.
There are different ways for proving inapproximability. A well–known technique consists in showing that the existence of an approximation algorithm for a problem would decide in polynomial time some NP–complete problem thus contradicting the common belief that $\mathcal{P} \neq \mathcal{NP}$.

1.7 Some variants of the 0–1 Knapsack Problem

In the following, we mention some classical variants of KP. An exhaustive discussion of these problems is far beyond the scope of this section. The reader may refer to the literature for further details on these variants and on other knapsack–like problems. The literature review of the generalizations of KP tackled in the thesis is given in the corresponding chapters.

A natural extension of KP is the *Bounded Knapsack Problem* (BKP), in which many copies of each item are available. Formally, if we denote by b_j the number of copies of item j, the ILP formulation of the problem corresponds to model (KP) where constraint (1.3) is replaced by:

$$x_j \leq b_j, x_j \in \mathbb{N}_0 \qquad j = 1, \ldots, n \tag{1.39}$$

The *Bounded Knapsack Problem* reflects situations for example in cargo loading where different quantities of the same product can be transported. A variant is the *Unbounded Knapsack Problem* (UKP) where each item has an unlimited number of copies.

A famous special case of KP is the *Subset Sum Problem* (SSP), in which the profits of the items are equal to their weights, i.e. $p_j = w_j$. The optimal solution of this problem consists in filling the knapsack capacity as much as possible. The *Subset Sum Problem* has been intensively studied because of its numerous applications in different combinatorial optimization problems.

Many real–life applications require the modeling of more than a capacity constraints. For example, items may have to be packed by considering limits also on their volumes and/or other features. The resulting generalization of KP is the *multidimensional knapsack problem* (MKP) which is another problem strongly investigated in the literature. The problem can be formulated by replacing the capacity constraint (1.2) in model (KP) by the series of inequalities

$$\sum_{j=1}^{n} w_{ij} x_j \leq c_i \qquad i = 1, \ldots, d \tag{1.40}$$

where w_{ij} and c_i denote the weights of the items j and the capacity value for the dimension $i = 1, \ldots, d$.

Another challenging variant of KP is the *quadratic knapsack problem* (QKP) in which the total profit obtained by a solution reflects how well the items fit together. Denoting by p_{jj} the profit of item j and by p_{ij} the additional profit (or cost) if item j is packed together with another item i, QKP is formulated by replacing the objective function of model (KP) by:

$$\text{maximize} \quad \sum_{i=1}^{n} \sum_{j=1}^{n} p_{ij} x_i x_j \tag{1.41}$$

A further variant is the Multiple Choice Knapsack Problem (MCKP). The items belong to disjoint classes and MCKP requires that exactly one item from each class has to be placed into the knapsack. Let us denote m disjoint classes of items by $N_1, \ldots, N_m$. Each item $j \in N_i$ has a weight w_{ij} and a profit p_{ij}. MCKP can be expressed by the following model

$$\text{maximize} \quad \sum_{i=1}^{m} \sum_{j \in N_i} p_{ij} x_{ij} \tag{1.42}$$

$$\text{subject to} \quad \sum_{i=1}^{m} \sum_{j \in N_i} w_{ij} x_{ij} \leq c \tag{1.43}$$

$$\sum_{j \in N_i} x_{ij} = 1 \qquad i = 1, \ldots, m, \tag{1.44}$$

$$x_{ij} \in \{0, 1\} \qquad i = 1, \ldots, m, \quad j \in N_i \tag{1.45}$$

where constraint (1.44) ensures that exactly one item is selected from the corresponding class. In order to guarantee feasible and not trivial solutions, it is commonly assumed that:

$$\sum_{i=1}^{m} \min_{j \in N_i} \{w_{ij}\} \leq c < \sum_{i=1}^{m} \max_{j \in N_i} \{w_{ij}\} \tag{1.46}$$

The 0–1 Knapsack Problem with Setups

2.1 Introduction

The 0–1 Knapsack Problem with Setups (KPS - originally introduced in [16]) can be seen as a generalization of KP where items belong to disjoint families (or classes) and can be selected only if the corresponding family is activated. The selection of a family involves setup costs and resource consumptions thus affecting both the objective function and the capacity constraint. KPS has many applications of interest such as make–to–order production contexts, cargo loading and product category management among others and more generally for resource allocation problems involving classes of elements (see, e.g., [17]).

Another application of KPS comes from the smart-home paradigm where the goal of efficiently managing the energy consumption in buildings is a strong commitment (see Project FLEXMETER funded by the European Commission under H2020 [1]). Here energy providers are requested to manage peak demands while satisfying an aggregated demand curve in order to avoid blackouts due to high peak demands. In this context, it may be required to shut down several home appliances whenever a Demand Response event for overall exceeding energy consumption is identified. This corresponds to select the best appliances to be shut down, by taking into account their relevance and their energy consumption, while also minimizing the houses involved in this shut down. Here, the families of items are the houses that we do not want to shut down and the items are their appliances. Thus, this is another practical application of KPS.

In [16], the authors consider the case with setup costs and profits of items being either positive or negative. A pseudopolynomial time dynamic programming approach and a two–phase enumerative scheme are proposed.

Considering the pseudo–polynomial time algorithm of [16] and the fact that KP is a special case of KPS, namely in the case where there is only one family, we can state that KPS is weakly NP–hard. In [89], a branch and bound algorithm is proposed for KPS. The algorithm turns out to solve instances with up to 10000 variables. Nonetheless the approach does not solve several large instances due to memory overflow. The current state-of-the-art exact approach for KPS is the one reported in [17] where an improved dynamic programming procedure is proposed. The procedure favorably compares to the commercial solver CPLEX 12.5 since it solves to optimality instances with up to 10000 items which turn out to be harder than the ones proposed in [89]. Further references can be found in [17].

Also a number of problems closely related to KPS were treated in the literature. In [14], a metaheuristic–based algorithm (cross entropy) is proposed in order to deal with KPS with more than one copy per item (cf. the Bounded Knapsack Problem). In [4], a variant of KPS with fractional items is considered and the authors present both heuristic methods and an exact algorithm based on cross decomposition techniques. In [3], the special case of KPS with no setup capacity consumptions but only setup costs is considered. For this so-called fixed–charge knapsack problem the author proposes both heuristic procedures and an exact branch and bound algorithm. A valuable overview of the literature of various KPS variants is provided in [65], which also devises a branch and bound scheme.

In this chapter, we propose two solution approaches for KPS. At first, we introduce an exact enumerative approach for KPS relying on an effective exploration of the solution space which exploits the partitioning of the variables set into two levels and requires the solution of several ILP models that show up to be easy to solve in practice. While the idea of approaching a combinatorial optimization problem by solving related simpler ILP formulations was already done in heuristic procedures (see, for instance, [24] for the closest string problem), here we do it within an exact approach.

Then, we introduce a new improved dynamic programming algorithm motivated by the connection of KPS to a knapsack problem with precedence constraints. The main aim of the new dynamic programming is to derive approximation results for KPS. At the same time, this pseudo–polynomial algorithm can be stated with less involved notation and turns out to outperform the recent dynamic programming approach by [17]. Moreover, it performs comparably to the proposed exact enumerative approach but avoids the use of an ILP-solver.

After discussing our solution approaches, we provide further insights into KPS and derive a number of approximation results. More precisely, we show that no polynomial approximation algorithm can exist for the problem in the general case unless $\mathcal{P} = \mathcal{NP}$ and we investigate several conditions for deriving fully polynomial time approximation schemes (FPTASs). Thus, we make progress in characterizing the borderline between non-approximability and existence of approximation schemes.

The research contribution of this chapter resulted in papers published ([25]) or accepted for publication ([72]) in international journals. A part of the contents has been presented at the international conference EURO 2015 and at the national conference AIRO 2014.
The chapter is organized as follows. In Section 2.2, the linear programming formulation of the problem is briefly described. We present the exact enumerative approach in Section 2.3 and the new dynamic programming algorithm in Section 2.4. In Section 2.5 computational results are discussed. Approximation results are discussed in Section 2.6.

2.2 Notation and problem formulation

In KPS a set of N families of items is given together with a knapsack with capacity b. Each family $i \in \{1, \dots, N\}$ has n_i items, a non–negative setup cost represented by an integer f_i and a non–negative setup capacity consumption denoted by an integer d_i. The total number of items is denoted by $n := \sum_{i=1}^{N} n_i$. Each item $j \in \{1, \dots, n_i\}$ of a family i has a non–negative integer profit p_{ij} and a non–negative integer capacity consumption w_{ij}. The problem calls for

maximizing the total profit of the selected items minus the fixed costs of the selected families without exceeding the knapsack capacity b. W.l.o.g. we can assume that $w_{ij} + d_i \leq b$ for all items. Any item violating this condition can be removed from the problem since it can never be part of a feasible solution. To derive an ILP-formulation we associate with each item j of family i a binary variable x_{ij} such that $x_{ij} = 1$ iff item j of family i is placed in the knapsack. To each family i we associate a binary variable y_i such that $y_i = 1$ iff family i is activated. The following ILP formulation of KPS (denoted KPS_1) holds.

KPS_1:

$$\text{maximize} \quad \sum_{i=1}^{N} \sum_{j=1}^{n_i} p_{ij} x_{ij} - \sum_{i=1}^{N} f_i y_i \tag{2.1}$$

$$\text{subject to} \quad \sum_{i=1}^{N} \sum_{j=1}^{n_i} w_{ij} x_{ij} + \sum_{i=1}^{N} d_i y_i \leq b \tag{2.2}$$

$$x_{ij} \leq y_i \qquad j = 1, \ldots, n_i, \quad i = 1, \ldots, N \tag{2.3}$$

$$x_{ij} \in \{0,1\} \qquad j = 1, \ldots, n_i, \quad i = 1, \ldots, N \tag{2.4}$$

$$y_i \in \{0,1\} \qquad i = 1, \ldots, N \tag{2.5}$$

The objective function (2.1) maximizes the sum of the profits of the selected items minus the costs induced by the activated families; the capacity constraint (2.2) guarantees that the sum of weights for selected items and families does not exceed the capacity value b; constraints (2.3) ensure that an item can be chosen if and only if the corresponding family is activated; finally constraints (2.4), (2.5) indicate that all variables are binary.

2.3 An exact enumerative solution approach

2.3.1 Rationale and preliminaries

Let us denote by KPS^{LP} the linear relaxation of KPS_1 where both integrality constraints (2.4) and (2.5) are replaced by the inclusion in $[0,1]$. It is known [89] that there exists at least one optimal solution of KPS^{LP} where there is at most one fractional variable y_i while there are typically many fractional variables

x_{ij}. We recall the special structure of KPS^{LP} sketched by [89] in Section 2.6.1. Anyhow, as an example we tested an instance from [17] with 10000 variables and 30 families: the optimal continuous solution has one fractional variable y_i and 330 fractional variables x_{ij}. Then, branching on any fractional x_{ij} always induced continuous solutions with more than 300 fractional x_j and one fractional y_i (often different from the one related to the original problem). Besides, branching on the fractional y_i, induced again fractional continuous solutions (always more than 300 fractional x_{ij} and another fractional y_i). This, presumably, is the main reason for which a standard ILP solver runs already into difficulties on several instances of KPS_1 with 1000 items (see Section 2.5). Our approach instead aims to exploit the structure of KPS, where the set of variables is partitioned into two levels, variables y_i (first level variables) and variables x_{ij} (second level variables). The practical hardness of the problem comes from these two sets of variables that must be properly combined to reach an optimal solution. At the same time, once the families are chosen, KPS boils down to a standard KP, which is in practice well handled by nowadays ILP solvers and specialized algorithms. Note that the idea of using approaches based on the repeated solution of NP–hard sub–problems is not new. For instance, in [2], the famous shifting bottleneck procedure for the job shop problem was based on the repeated solution of a single machine problem with release times and tails that, although being NP–hard in the strong sense, is well solved in practice by the exact algorithm in [13]. Here, as the selection of the families induces problems that are tractable in practice, we focus on an efficient exploration of the solution space defined by the first level variables. In particular, we propose an exact enumerative approach, hereafter denoted as $FLEA$ (for First Level variables Exploration based Approach), based on the idea of limiting the range on the number of families that may lead to an optimal solution and seeking for solutions within this range. Three main steps are involved. In the first step an initial feasible solution is computed and a standard variable fixing procedure is applied through the reduced costs of the non–basic variables in the optimal solution of the continuous relaxation of the problem. The second step concerns the detection of the range of possible optimal number of families. This leads to the identification of sub–problems that are tackled in the third phase. We use the ILP solver (CPLEX 12.5)

whenever the solution of an ILP model is required by our approach. In the following, we describe the three steps of $FLEA$ algorithm whose pseudo code is presented below.

2.3.2 Initial feasible solution computation and variables fixing

We start by considering KPS^{LP} where, in addition, we require the sum of the selected families to be integer. Thus, we get the following model (denoted by KPS_2).

KPS_2:

$$\text{maximize} \quad \sum_{i=1}^{N}\sum_{j=1}^{n_i} p_{ij}x_{ij} - \sum_{i=1}^{N} f_i y_i \tag{2.6}$$

$$\text{subject to} \quad (2.2), (2.3)$$

$$\sum_{i=1}^{N} y_i = k \tag{2.7}$$

$$0 \leq x_{ij} \leq 1 \quad \forall j = 1,\ldots,n_i, \quad \forall i = 1,\ldots,N \tag{2.8}$$

$$0 \leq y_i \leq 1 \quad \forall i = 1,\ldots,N \tag{2.9}$$

$$k \in \mathcal{N} \tag{2.10}$$

Here, the integrality constraints on variables x_{ij} and y_i of KPS_1 are replaced by the inclusion in [0,1] while constraint (2.7) forces the sum of the families to take an integer value through the integer variable k. The optimal solution of this problem gives an upper bound on the KPS optimum. Moreover, the optimal value of k, denoted by k^*, provides a first guess on the total number of families to include in a solution. Then, we consider again model KPS_1 with the additional constraint that the number of families to activate is fixed to a value S and we remove the integrality constraints on variables x_{ij} only. Correspondingly, we get hereafter the following model (denoted by KPS_3).

KPS_3:

$$\text{maximize} \quad \sum_{i=1}^{N}\sum_{j=1}^{n_i} p_{ij}x_{ij} - \sum_{i=1}^{N} f_i y_i \tag{2.11}$$

$$\text{subject to} \quad (2.2), (2.3)$$

$$\sum_{i=1}^{N} y_i = S \tag{2.12}$$

$$0 \leq x_{ij} \leq 1 \quad \forall j = 1, \ldots, n_i \quad \forall i = 1, \ldots, N \tag{2.13}$$

$$y_i \in \{0,1\} \quad \forall i = 1, \ldots, N \tag{2.14}$$

We may expect that problem KPS_3 is easy to solve as only the y_i variables are binary and the number of families is relatively limited. Further, the solution space is restricted to the hyperplane representing the sum S in constraint (2.12). This argument shows up to hold in practice. We first solve KPS_3 by setting $S = k^*$. The optimal solution provides a feasible combination of y_i, denoted by the 0–1 vector y'.

If we consider the combination y' in KPS_1, we induce a KP with the capacity constraint and objective function modified according to the setups of the families. For the sake of simplicity, hereafter we refer to

$$KPS_1(y') = KPS_1 \cap \left(y_i = y'_i\right) \quad \forall i = 1, \ldots, N \tag{2.15}$$

as the standard knapsack problem related to any specific combination of families encoded by vector y'.

Solving $KPS_1(y')$ provides a first feasible solution for KPS. Let us denote the corresponding solution value by LB'. This is sketched in lines 2-6 of $FLEA$ pseudo code.

Then, we solve KPS^{LP}. We denote the optimal value of KPS^{LP} by z^{LP} and the optimal values of variables x_{ij} and y_i by x_{ij}^{LP} and y_i^{LP} respectively. Let $r_{x_{ij}}$ and r_{y_i} be the reduced costs of non basic variables in the optimal solution of KPS^{LP}. We then apply standard variable-fixing techniques from Integer Linear Programming. It is well known (see, for instance, [22]) that, if the gap

between the best feasible solution available and the optimal solution value of the continuous relaxation solution is not greater than the absolute value of a non basic variable reduced cost, then the related variable can be fixed to its value in the continuous relaxation solution. Correspondingly, we evaluate the reduced costs of all non basic variables in the optimal solution of KPS^{LP}. Then, the following constraints are added to the models (lines 7–8 of $FLEA$):

$$\forall i,j : |r_{x_{ij}}| \geq z^{LP} - LB', \quad x_{ij} = x_{ij}^{LP}; \tag{2.16}$$

$$\forall i : |r_{y_i}| \geq z^{LP} - LB', \quad y_i = y_i^{LP}. \tag{2.17}$$

2.3.3 Identifying the relevant sums of the families

Given the first solution LB', the number of families in an optimal solution can be bounded straightforwardly by solving two continuous problems. More precisely, we minimize and maximize $\sum y_i$ subject to constraints (2.2), (2.3) and to an additional constraint ensuring that the total profit must be strictly greater than the current solution value. The corresponding ILP formulations (denoted by KPS_{min} and KPS_{max} respectively) are as follows.

KPS_{min} (KPS_{max}):

$$\min\ (\max) \sum_{i=1}^{N} y_i \tag{2.18}$$

$$\text{subject to} \quad (2.2), (2.3)$$

$$\sum_{i=1}^{N}\sum_{j=1}^{n_i} p_{ij}x_{ij} - \sum_{i=1}^{N} f_i y_i \geq LB' + 1 \tag{2.19}$$

$$0 \leq x_{ij} \leq 1 \quad \forall j = 1,\ldots,n_i, \quad \forall i = 1,\ldots,N \tag{2.20}$$

$$0 \leq y_i \leq 1 \quad \forall i = 1,\ldots,N \tag{2.21}$$

Ceiling and flooring the optimal solution values of the above problems yield the lower and upper bound on the number of families possibly leading to an optimal solution of KPS. We denote these bounds by S_{min} and S_{max}

respectively . The second step of our approach is summarized in lines 9–12 of $FLEA$ algorithm.

2.3.4 Solving sub–problems

The third step consists in exploring sub–problems for the possible values of S in the range $[S_{min}, S_{max}]$ (for–loop in lines 13–24 of $FLEA$).
For each sub–problem we first solve KPS_3 and find a combination of families $\bar{y}$ as in Section *2.3.2* (lines 14–15 of $FLEA$). Then we solve $KPS_1(\bar{y})$ and if its optimal value is greater than the current best feasible solution value, we update the latter one (lines 17–20 of $FLEA$). We solve to optimality a KP, but indeed $\bar{y}$ is not guaranteed to be optimal for KPS_1. So we search for another possible combination of y_i within the sub–problem by adding to KPS_3 the constraint:

$$\sum_{i=1}^{N} \bar{y}_i y_i \leq S - 1 \tag{2.22}$$

This is a cut in the solution space imposing that at least one of the families of the previous combination must be discarded. We solve KPS_3 with one more constraint and apply the same procedure until the upper bound provided by solving KPS_3 is not better than the current best solution value or the problem becomes infeasible (while–loop in lines 16–23 of $FLEA$). We note that KPS_3 can turn out to be difficult to solve when further constraints on variables y_i are added. Nevertheless, additional cuts showed up to be reasonably limited. Once all sub–problems have been investigated, an optimal solution of KPS is obtained.

We note that steps 1–12 in $FLEA$ algorithm are executed only once, requiring the solution of problems KPS_2, KPS_3, $KPS_1(y')$, KPS^{LP}, KPS_{min} and KPS_{max} and the variable fixing (running in $O(\sum_{i=1}^{N} n_i)$ time) induced by constraints (2.16, 2.17). Also, the for–loop in lines 13–24 is repeated $[S_{max} - S_{min} + 1] = O(N)$ times where at each iteration: (1) KPS_3 is solved once and (2) the while-loop is executed in lines 16–23 requiring first the solution of $KPS_1(\bar{y})$ and then the solution of KPS_3 until $\overline{UB} \leq Best$. Thus

Algorithm 2 *FLEA*

Input: KPS instance.
▷ Find first solution and fix variables
$k^* \leftarrow$ solve KPS_2;
$S \leftarrow k^*$;
$(UB', y') \leftarrow$ solve KPS_3;
$LB' \leftarrow$ solve $KPS_1(y')$;
$Best = LB'$;
Solve KPS^{LP};
Apply (2.16, 2.17) and fix variables;
$z_{min} \leftarrow$ solve KPS_{min}; ▷ Identify the range of families
$z_{max} \leftarrow$ solve KPS_{max};
$S_{min} = \lceil z_{min} \rceil$;
$S_{max} = \lfloor z_{max} \rfloor$;
for all s in $[S_{min}, S_{max}]$ **do** ▷ Solve sub–problems
    $S = s$;
    $(\overline{UB}, \bar{y}) \leftarrow$ solve KPS_3;
    while $\overline{UB} \geq Best + 1$ **do**
        $\overline{LB} \leftarrow$ solve $KPS_1(\bar{y})$;
        if $\overline{LB} > Best$ **then**
            $Best = \overline{LB}$;
        end if
        add $(\sum_{i=1}^{N} \bar{y}_i y_i \leq S - 1)$ to KPS_3;
        $(\overline{UB}, \bar{y}) \leftarrow$ solve KPS_3;
    end while
end for
return $Best$;

the bottleneck of the algorithm is indeed the total number of times the while–loop is executed which could be potentially large but computational testing indicates that this number is very small in practice (never greater than 33 for instances with up to 100000 items).

2.4 A new dynamic programming algorithm

The second algorithm we propose is motivated by a dynamic programming scheme for a special case of the *precedence constraint knapsack problem* (PCKP) which is a variant of the standard knapsack problem. In this section we briefly recall PCKP and its relation to KPS. Then our dynamic programming algorithm is presented.
We remark that the idea of our dynamic program shares structural elements with the dynamic programming approach devised in [44]. However, our main goal is to offer a very simple and viable alternative to the algorithms for KPS available in the literature without relying on the use of an ILP solver.
Moreover, we exploit our dynamic programming for deriving the approximation results in Section 2.6. For this purpose, we perform dynamic programming by profits although we could just as well run the dynamic program over all weight values.

2.4.1 The Precedence Constraint Knapsack Problem (PCKP)

The *precedence constraint knapsack problem* (PKCP) or *partially ordered knapsack problem* (see e.g. [47, ch. 13.2] for an overview) is a generalization of KP which imposes a partial order on the items. This means that for each item there is a (possibly empty) set of predecessors which have to be packed into the knapsack if the item is packed. Formally, we are given a directed acyclic graph $G = (V, A)$ where the vertices of V correspond to the items and the existence of an arc $(i, j) \in A$ means that item j can only be part of a solution if also item i is in the solution.

Note that it makes sense to permit also negative profit values in PKCP since due to the precedence constraints also items with negative profits may be part of an optimal solution. PKCP is NP–hard in the strong sense but it is solvable in pseudo–polynomial time on graphs like *out–trees* and *in–trees*. An out–tree is a directed tree with a distinguished vertex called the *root* such that there is a directed path (which is unique) from the root to every vertex in the tree.

KPS can be easily modeled as a PCKP on an out-tree as follows. First, introduce as a root a dummy item with zero weight and profit which has directed arcs to N vertices each of which representing one family. An item of PCKP corresponding to such a *family vertex* carries the setup capacity consumption of the associated family as its weight and the setup cost as negative profit. The actual knapsack items of KPS are inserted in PCKP as leaves of the tree with an incoming arc emanating from the respective family vertex. Thus, the selection of an item requires the activation of the corresponding family.

A dynamic programming algorithm for PCKP for out-trees, called the *left–right approach*, is due to [44]. The main idea of this profit-based algorithm is a suitable ordering of sub–problems identified by sub–trees which corresponds to the structure of the dynamic programming recursion. The computational complexity of this approach is in $O(nUB)$, with UB indicating an upper bound on the objective function value. A variant of this algorithm employing dynamic programming by weights was given by [18]. Its running time is in $O(nb)$.

2.4.2 Dynamic Programming for KPS

Let us denote by KPS(i,j) the sub–problem induced by the first $i-1$ families, all their items and the first j items of family i. Also, indicate by $r(i,\bar{p})$ the smallest weight for which a solution of KPS(i,n_i) with a total profit at least $\bar{p}$ exists, namely:

$$r(i,\bar{p}) := \min_{S}\{w(S) \mid S \text{ is a solution of KPS}(i,n_i) \text{ and } p(S) \geq \bar{p}\}$$

Here $w(S)$ and $p(S)$ represent the sum of the weights and profits of the items and families included in a solution set S. The minimum over the empty set

will be written as ∞. As a restricted variant of $r(i,\bar{p})$ where a certain family i is forced to be activated we define for a triple $(i,j,\bar{p})$:

$$z(i,j,\bar{p}) := \min_{S'}\{w(S') \mid S' \text{ is a solution of KPS}(i,j) \text{ with family i activated and } p(S') \geq \bar{p}\}$$

In case that no feasible solution exists for $r(\cdot)$ or $z(\cdot)$ we will set the corresponding entry to ∞. Note that not packing any item can also be a feasible solution with profit and weight equal to 0.

The idea is to analyze the families one by one according to a proper ordering of the sub–problems. More precisely, we first consider the sub–problem with the first family included without any of its items by calculating $z(1,0,\bar{p})$. After that we consider the possible selection of the items of the family one after the other. The analysis of the contribution of the first family is completed by computing $r(1,\bar{p})$. Then we proceed with the other families until $r(N,\bar{p})$ is determined. In each iteration the smallest weight to reach a certain profit level $\bar{p}$ with the first i families is stored in $r(i,\bar{p})$ and then this information is used to analyze the contribution of the next family.

The related approach for PCKP assumes positive profit values. For KPS we have to consider also the cases with negative profits to cover the activation costs of a family. If a family ends up with a negative total profit, its items would not be included at all. Thus, the minimal profit value p_0 the algorithm has to consider is bounded by the negative maximum setup costs.

$$p_0 = -\max\{f_i \mid i = 1,\ldots,N\} \tag{2.23}$$

As an upper bound UB for the objective function value we could round down the optimum value of the continuous relaxation of KPS. We remark that the gap between the linear relaxation and the optimal solution value can be arbitrarily large as we will see in Section 2.6.1, nevertheless the bound should be reasonably close to the optimal value in practice. A possible alternative is to consider the trivial upper bound given by neglecting the capacity constraint and selecting all items of all families: $\sum_{i=1}^{N}\sum_{j=1}^{n_i} p_{ij} - \sum_{i=1}^{N} f_i$.

In an initialization step we compute $r(0,\bar{p})$, namely the case where no family is activated and no positive profit can be obtained:

1. for each $(p_0 \leq \bar{p} \leq UB)$

$$r(0,\bar{p}) = \begin{cases} 0 & \text{if } \bar{p} \leq 0, \\ \infty & \text{otherwise.} \end{cases}$$

After this initialization step, the following recursions are iteratively applied for each family $(1 \leq i \leq N,\ 1 \leq j \leq n_i)$:

2.

$$z(i,0,\bar{p}) = \begin{cases} r(i-1,\bar{p}+f_i)+d_i & \text{if } r(i-1,\bar{p}+f_i)+d_i \leq b \\ & \bar{p}+f_i \leq UB, \\ \infty & \text{otherwise.} \end{cases}$$

3.

$$z(i,j,\bar{p}) = \begin{cases} \min\{z(i,j-1,\bar{p}), z(i,j-1,k)+w_{ij}\} & \text{if } z(i,j-1,k)+w_{ij} \leq b \\ & (k = \max\{p_0, \bar{p}-p_{ij}\}), \\ z(i,j-1,\bar{p}) & \text{otherwise.} \end{cases}$$

4.

$$r(i,\bar{p}) = \min\{r(i-1,\bar{p}), z(i,n_i,\bar{p})\}$$

As far as Rule 2 is concerned, by definition family i must be contained in the solution by contributing to the objective only with its setup cost f_i. Thus Rule 2 evaluates if a profit $\bar{p}$ can be obtained by checking whether the sum of the weight to get a profit $\bar{p}+f_i$ with the first $i-1$ families $(r(i-1,\bar{p}+f_i))$ and the setup weight d_i does exceed b or not. Indeed a value $\bar{p}+f_i$ exceeding UB would lead to an infeasible solution and it is not considered. Likewise Rule 3 evaluates the insertion into the knapsack of the item represented by x_{ij} to get a profit $\bar{p}$. Clearly we must have $\bar{p}-p_{ij} \geq p_0$ in that case. The classical knapsack recursion introduced in Rule 4 simply establishes whether the family i is put into the knapsack or not. The optimal solution for KPS is determined by taking the maximum value $\bar{p}$ for which $r(N,\bar{p}) < \infty$.

To bound the pseudo–polynomial running time of this algorithm for KPS it suffices to observe that going through all families the array $z(\cdot)$ is evaluated only once for each item in combination with every profit value while $r(\cdot)$ is evaluated only for every family and every profit value. This yields a trivial $O(n(UB+|p_0|))$ time bound.

In practice, we can obtain a considerable speed-up by systematically considering upper bounds lower than UB for the sub–problems at hand. This will lead to drastic improvements, especially in the early stages of the algorithm where a limited number of families is involved.

It is not difficult to modify the above recursions for dynamic programming by weights where each array entry represents a solution of maximum profit for a certain weight bound $\bar{w} = 1,\ldots,b$. Without going into details we just state that the resulting time complexity is $O(n\,b)$.

From a practical perspective it is also interesting to analyze different choices regarding the sequence of families and items. In our computational tests we consider – somehow counterintuitively – first the *less efficient* families and the less efficient items within each family. As usual, the efficiency of an item is the ratio of profit over weight. For each family, the efficiency is defined as the ratio between the sum of profits of the items minus the setup cost and the sum of the weights (including setup capacity consumption). This choice aims to obtain small upper bounds in the first steps of the dynamic program thus yielding savings in the total running time.

Historically, space requirements were seen as a major obstacle for applying dynamic programming algorithms to large problem instances. Although today's hardware specifications diminish this issue, it is still relevant to be considered also from a theoretical point of view. It is easy to see that for evaluating $z(i,0,\bar{p})$ and $r(i,\bar{p})$ only the values of $r(i-1,\cdot)$ are required. Hence, we can overwrite the values of $r(i-2,\cdot)$ when we start the iteration for family i and restrict the space requirement for array r to $O(UB+|p_0|)$. The same argument applies for z: Entries $z(i,j,\bar{p})$ require only $z(i,j-1,\cdot)$ for $j = 1,\ldots,n_i$. Thus the space bound of $O(UB+|p_0|)$ suffices also for z.

Another – often overlooked – aspect concerns the storage of the actual solution sets of packed items. Recall that each dynamic programming entry should contain not only a weight value but also the associated item set (cf. [47, ch. 2.3]). This would increase the space requirements by factor of n (or $\log n$, if a bit representation of item sets is employed). However, we can adapt the general recursive storage reduction principle from [70] (see also [47, ch. 3.3]) to keep the space complexity of $O(n+UB+|p_0|)$ without increasing the running time of $O(n(UB+|p_0|))$ but still reporting the optimal solution set. Note that this storage reduction scheme requires an equipartition of the item set and the independent solution of the two resulting sub–problems in every iteration. This is in general impossible for the items of KPS which are structured into families. However, for the correctness of the recursive argument of [70] it suffices to construct in each iteration a partitioning of the items into two sets, each of them containing at least a constant fraction of the current item set. Without going into details, we mention that we can realize a partitioning into two subsets each of them containing at least a quarter of the current item set by either partitioning complete families or partitioning only the items of the largest family if it contains at least one half of the current item set.

2.5 Computational results

We first present an extensive computational analysis of $FLEA$ algorithm. More precisely, we compared the proposed approach with the algorithms available in the literature and we evaluated its scalability on larger instances never tested before. Besides, we report a direct comparison of our dynamic programming with the one laid out in [17].

For what concerns $FLEA$, all tests were conducted on an Intel i5 CPU @ 3.3 GHz with 4 GB of RAM. We used the ILP solver CPLEX 12.5 and the code was implemented in the C++ programming language. The parameters of CPLEX 12.5 were set to their default values.

We generated the instances according to the scheme provided in [89]. In addition, we also considered the instances available in [17]. In the scheme provided in [89], the number of families N is 50 and 100. The cardinalities n_i

of the families are integers uniformly distributed in the ranges [40, 60] and [90, 110]. Setup costs and weights are given by

$$f_i = e_1 \left(\sum_{j=1}^{n_i} p_{ij} \right) \tag{2.24}$$

$$d_i = e_2 \left(\sum_{j=1}^{n_i} w_{ij} \right) \tag{2.25}$$

where e_1 and e_2 are uniformly distributed in the intervals [0.05, 0.15], [0.15, 0.25], [0.25, 0.35] and [0.35, 0.45]. In the uncorrelated instances, both the items weights w_{ij} and profits p_{ij} are integer randomly distributed in the range [10, 10000]. In the correlated instances the profits are integer randomly distributed in the range [w_{ij}-1000, w_{ij}+1000], but if the profits are less than 10, then they range in the interval [10, 100]. The capacity b is an integer randomly distributed in the range $\left[0.4 \left(\sum_{i=1}^{N} \sum_{j=1}^{n_i} w_{ij} \right), 0.6 \left(\sum_{i=1}^{N} \sum_{j=1}^{n_i} w_{ij} \right) \right]$.

			CPLEX 12.5			Algorithm *FLEA*			
N	n_i	*Setup*	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	Max #Sub–pb	#Opt
50	[40-60]	[0.05-0.15]	0.31	0.38	10	0.50	0.64	1	10
		[0.15-0.25]	0.34	0.42	10	0.53	0.62	1	10
		[0.25-0.35]	0.45	0.59	10	0.57	0.62	1	10
		[0.35-0.45]	0.35	0.56	10	0.45	0.57	1	10
50	[90-110]	[0.05-0.15]	0.49	0.66	10	0.71	0.91	1	10
		[0.15-0.25]	0.73	0.97	10	0.90	1.03	1	10
		[0.25-0.35]	2.15	10.41	10	0.96	1.15	1	10
		[0.35-0.45]	1.12	2.68	10	0.86	1.26	1	10
100	[40-60]	[0.05-0.15]	0.45	0.66	10	0.67	0.88	1	10
		[0.15-0.25]	0.69	0.91	10	0.82	1.00	1	10
		[0.25-0.35]	0.65	1.03	10	0.80	1.17	1	10
		[0.35-0.45]	0.65	0.89	10	0.76	0.97	1	10
100	[90-110]	[0.05-0.15]	0.98	1.33	10	1.16	1.58	1	10
		[0.15-0.25]	1.86	3.25	10	1.73	2.22	1	10
		[0.25-0.35]	1.35	2.15	10	1.49	1.75	1	10
		[0.35-0.45]	1.33	2.08	10	1.52	2.41	1	10

Table 2.1 KPS uncorrelated instances with w_{ij} and p_{ij} in [10, 10000]: time (s) and number of optima.

			CPLEX 12.5			Algorithm *FLEA*			
N	n_i	*Setup*	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	Max #Sub–pb	#Opt
50	[40-60]	[0.05-0.15]	1.02	2.53	10	0.72	1.14	2	10
		[0.15-0.25]	0.67	0.97	10	0.59	0.89	2	10
		[0.25-0.35]	0.63	1.45	10	0.61	0.78	1	10
		[0.35-0.45]	0.96	2.62	10	0.65	0.81	2	10
50	[90-110]	[0.05-0.15]	3.53	13.14	10	0.99	1.31	1	10
		[0.15-0.25]	7.65	22.17	10	1.18	1.47	1	10
		[0.25-0.35]	3.41	10.87	10	1.15	1.42	1	10
		[0.35-0.45]	9.46	39.08	10	1.27	1.81	1	10
100	[40-60]	[0.05-0.15]	1.51	5.40	10	0.81	1.08	1	10
		[0.15-0.25]	1.95	7.13	10	0.98	1.31	1	10
		[0.25-0.35]	1.01	2.48	10	1.04	1.37	2	10
		[0.35-0.45]	1.37	2.50	10	1.17	1.76	2	10
100	[90-110]	[0.05-0.15]	11.15	71.98	10	1.93	2.59	1	10
		[0.15-0.25]	31.14	173.21	10	2.00	2.50	1	10
		[0.25-0.35]	71.22	652.02	10	2.30	4.57	2	10
		[0.35-0.45]	15.32	42.56	10	2.18	3.40	2	10

Table 2.2 KPS correlated instances with w_{ij} in $[10, 10000]$ and p_{ij} in $[w_{ij} - 1000, w_{ij} + 1000]$: time (s) and number of optima.

We compared the solutions reached by CPLEX 12.5 running on KPS_1 to the solutions obtained with $FLEA$ over 10 instances within each category. The results are reported in Tables 2.1 and 2.2 in terms of average and maximum CPU time and number of optima reached within a time limit of 1200 seconds. We also report the maximum number of the relevant sub–problems, that is $S_{max} - S_{min} + 1$, identified by the proposed algorithm.

Uncorrelated instances show up to be very easy to solve for both $FLEA$ and CPLEX 12.5. We remark that the same conclusion applies to the instances in [16], which are uncorrelated with positive or negative profits for the items and setup costs. As mentioned in [89], these instances are not difficult, since a preprocessing step reduces the problems size considerably.

For the correlated instances, CPLEX 12.5 solves to optimality all the instances but performs slightly worse. $FLEA$ reaches the optimum over all instances in no more than 5 seconds. We note that the method proposed in [89] requires significantly higher computational time and runs out-of-memory in several cases for similar correlated instances. So, even if we were not able

to obtain from the authors the instances reported in [89], we can reasonably expect our algorithm to outperform their approach.

We further tested a stronger correlation between the profits of the items and their weights. More precisely, we generated instances where w_{ij} is an integer uniformly distributed in the range [10, 100], while the profits of items are $p_{ij} = w_{ij} + 10$. The results are provided in Table 2.3.

			CPLEX 12.5			Algorithm *FLEA*			
N	n_i	*Setup*	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	Max #Sub–pb	#Opt
50	[40-60]	[0.05-0.15]	126.17	1200.00	9	1.80	4.91	2	10
		[0.15-0.25]	6.76	34.10	10	1.41	3.46	2	10
		[0.25-0.35]	4.25	10.06	10	1.20	2.43	3	10
		[0.35-0.45]	2.47	6.91	10	0.83	1.68	2	10
50	[90-110]	[0.05-0.15]	701.84	1200.00	5	2.13	4.52	2	10
		[0.15-0.25]	241.20	1200.00	9	2.70	9.86	2	10
		[0.25-0.35]	307.96	1200.00	9	2.10	4.84	2	10
		[0.35-0.45]	28.75	214.94	10	1.49	2.14	1	10
100	[40-60]	[0.05-0.15]	30.66	157.59	10	9.48	65.13	2	10
		[0.15-0.25]	18.85	100.34	10	3.83	13.62	2	10
		[0.25-0.35]	5.55	14.49	10	2.29	5.29	2	10
		[0.35-0.45]	9.65	24.24	10	2.69	7.00	3	10
100	[90-110]	[0.05-0.15]	498.44	1200.00	7	5.56	11.92	2	10
		[0.15-0.25]	197.00	1200.00	9	5.35	10.66	2	10
		[0.25-0.35]	267.26	1200.00	9	6.40	22.07	2	10
		[0.35-0.45]	188.75	1200.00	9	5.37	9.95	2	10

Table 2.3 KPS correlated instances with w_{ij} in $[10, 100]$ and $p_{ij} = w_{ij} + 10$: time (s) and number of optima.

These instances turned out to be harder to solve than the correlated instances in Table 2.2. A reasonable interpretation is that in [89] a weaker correlation is considered and weights vary in a much wider range ([10, 10000]), increasing the probability of having items much more profitable than others. Nevertheless *FLEA* still manages to handle all instances in very reasonable computational time, while CPLEX 12.5 is not capable of reaching all the optima. It is quite evident from our testing that one of the strengths of the proposed approach is the capacity of drastically limiting the number of sub-problems to be explored in the last step of the algorithm. A natural question that may arise is whether this last task can be accomplished just by letting an ILP solver tackle the sub-problems. It would indicate to what extent the procedure devised in the third step of our algorithm provides an effective contribution in solving the problem. We investigated this aspect by exploring the behavior of *FLEA* if CPLEX 12.5 is launched (with a time limit of 1200 seconds) on each of the

sub–problems in the third step of the method, that is the sub–problems of subsection 2.3.4. We denote as *FLEA(II)* this last version of the proposed approach.

We then compared the two versions of *FLEA* to the dynamic programming algorithm in [17] and to CPLEX 12.5 over a set of instances proposed in [17]. These instances involve a high level of correlation between profits and weights where w_{ij} is an integer uniformly distributed in the range [10,100] and $p_{ij} = w_{ij} + 10$. In Table 2.4, we report the performances of CPLEX 12.5, the two versions of *FLEA* and the dynamic programming algorithm proposed in [17]. The number of families varies from 5 to 30 and the total number of items n from 500 to 10000. Within each category, 10 instances were tested.

		CPLEX 12.5			Algorithm *FLEA*				Dynamic Progr. from [17]			*FLEA(II)* using CPLEX 12.5 for solving sub–problems		
N	n	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	Max #Sub–pb	#Opt	Average time (s)*	Max time (s)*	#Opt	Average time (s)	Max time (s)	#Opt
5	500	44.17	218.18	10	0.43	0.72	2	10	0.31	0.49	10	4.11	34.45	10
	1000	568.37	1200.00	7	0.51	0.66	1	10	0.92	1.06	10	192.25	1200.00	9
	2500	1106.42	1200.00	1	0.98	1.28	1	10	5.34	5.71	10	3.25	11.40	10
	5000	929.04	1200.00	3	1.57	1.84	1	10	20.81	21.52	10	126.53	1200.00	9
	10000	987.01	1200.00	2	3.03	3.67	1	10	83.93	85.19	10	17.98	58.52	10
10	500	71.73	423.98	10	0.46	0.64	2	10	1.50	11.32	10	1.11	5.83	10
	1000	1200.00	1200.00	0	0.47	0.67	2	10	1.27	1.38	10	0.61	0.91	10
	2500	1200.00	1200.00	0	0.84	1.01	1	10	7.33	7.72	10	121.28	1200.00	9
	5000	825.85	1200.00	4	1.47	1.62	1	10	29.18	30.52	10	633.70	1200.00	5
	10000	1200.00	1200.00	0	3.10	3.48	1	10	149.73	154.61	10	966.57	1200.00	2
20	500	382.23	1200.00	7	0.61	1.14	2	10	0.56	0.78	10	6.04	55.21	10
	1000	50.76	229.96	10	0.51	0.87	1	10	2.15	2.63	10	1.60	9.33	10
	2500	1200.00	1200.00	0	0.88	1.40	2	10	13.01	13.68	10	1.18	1.69	10
	5000	1054.83	1200.00	2	1.58	1.95	1	10	53.45	54.99	10	19.53	173.16	10
	10000	1200.00	1200.00	0	2.96	3.74	1	10	346.58	353.68	10	143.82	1200.00	9
30	500	237.75	1200.00	9	1.62	4.73	5	10	0.76	0.89	10	5.51	37.41	10
	1000	499.63	1200.00	8	0.87	1.95	2	10	3.32	3.63	10	2.44	17.33	10
	2500	1175.79	1200.00	1	0.99	1.25	2	10	19.58	20.20	10	3.39	13.21	10
	5000	380.40	1200.00	8	1.62	2.59	1	10	79.76	83.42	10	4.73	23.68	10
	10000	907.74	1200.00	5	4.82	8.07	2	10	526.61	549.03	10	37.36	249.21	10

Table 2.4 KPS benchmark instances (from [17]): time (s) and number of optima.

These instances involve a lower number of families and show up to be harder for CPLEX 12.5 than the previous ones. Nevertheless, even though CPLEX 12.5 runs out of time for most of the large instances, *FLEA* is able to find all optima with limited computational effort. The dynamic programming algorithm is capable of reaching all the optima as well. However the computational times are much larger and increase with the size of the instances. We remark that the tests in [17] were carried out on a slightly less performing machine (an asterisk is introduced in the table to point out that times refer to another machine, namely an Intel core TMi3 CPU @ 2.1 GHZ with 2GB of RAM). Anyhow given these results, it is very reasonable to assume that the differences in the performance would remain significant even if the algorithms were launched on

the same machine. We also note that the use of the ILP solver combined with the first two steps of the proposed approach, namely *FLEA(II)* algorithm, enhances the performance of CPLEX 12.5 launched on the ILP formulation of KPS. At the same time, this algorithmic variant turns out to be less performing than *FLEA* and not capable of reaching all the optima within the time limit. Then, we tested the scalability of *FLEA* on larger instances with $e_1 = e_2$ uniformly ranging in the interval [0.15, 0.25], w_{ij} integer uniformly distributed in the range [10,100] and $p_{ij} = w_{ij} + 10$, $b = 0.5 \left(\sum_{i=1}^{N} \sum_{j=1}^{n_i} w_{ij} \right)$ with the number of families and items up to 200 and 100000 respectively. The results are reported in Table 2.5.

		CPLEX 12.5			Algorithm *FLEA*			
N	n	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	Max #Sub–pb	#Opt
5	20000	380.64	1200.00	7	6.93	12.20	1	10
	50000	630.15	1200.00	5	55.49	97.38	1	10
	100000	752.66	1200.00	6	285.74	542.94	1	10
10	20000	374.49	1200.00	7	4.38	8.21	1	10
	50000	427.40	1200.00	7	27.14	52.92	1	10
	100000	731.80	1200.00	5	135.99	390.62	1	10
20	20000	1031.29	1200.00	2	6.35	9.16	1	10
	50000	1190.44	1200.00	1	26.98	43.01	1	10
	100000	1095.40	1200.00	1	102.92	231.63	1	10
30	20000	736.75	1200.00	4	9.40	14.49	1	10
	50000	749.91	1200.00	4	31.84	39.87	1	10
	100000	1092.11	1200.00	2	127.39	179.67	1	10
50	20000	685.84	1200.00	5	8.27	14.81	2	10
	50000	1196.78	1200.00	1	51.06	87.31	2	10
	100000	1139.12	1200.00	1	147.89	218.74	2	10
100	20000	750.62	1200.00	5	18.34	59.61	2	10
	50000	1116.64	1200.00	1	89.39	497.89	1	10
	100000	1090.48	1200.00	1	128.84	272.81	2	10
200	20000	367.69	1200.00	8	19.94	43.13	2	10
	50000	966.37	1200.00	3	105.39	284.50	1	10
	100000	1113.78	1200.00	1	163.02	359.18	2	10

Table 2.5 KPS larger instances: time (s) and number of optima.

We notice that *FLEA* effectively solves also these larger instances, requiring approximately 540 seconds for the worst-case instance with 100000 items. CPLEX 12.5 fails to reach the optimum within the time limit of 1200 seconds in more than 60% of the instances of Table 2.5.

These extensive computational experiments confirm the effectiveness of our algorithm, which strongly outperforms CPLEX 12.5 and the algorithms reported in the literature. *FLEA* is capable of solving to optimality all instances within the time limit.

Finally, we compared our dynamic programming with the dynamic programming described in [17] and with *FLEA*. We considered again the instances available in [17]. The code of our dynamic programming has been implemented in the C++ language under the Linux operating system. The results are reported in Table 2.6 in terms of average and maximum CPU time taken to reach the optimal solutions. A double asterisk in the table indicates that the times of our dynamic programming refer to another machine, i.e. an Intel i5 CPU @ 3.2 GHz with 16 GB of RAM.

		Algorithm *FLEA*		New Dynamic Programming		Dynamic Progr. from [17]	
N	n	Average time (s)	Max time (s)	Average time (s)**	Max time (s)**	Average time (s)*	Max time (s)*
5	500	0.43	0.72	0.02	0.02	0.31	0.49
	1000	0.51	0.66	0.07	0.08	0.92	1.06
	2500	0.98	1.28	0.54	0.56	5.34	5.71
	5000	1.57	1.84	2.00	2.16	20.81	21.52
	10000	3.03	3.67	8.67	8.98	83.93	85.19
10	500	0.46	0.64	0.01	0.02	1.50	11.32
	1000	0.47	0.67	0.07	0.08	1.27	1.38
	2500	0.84	1.01	0.49	0.50	7.33	7.72
	5000	1.47	1.62	1.79	1.84	29.18	30.52
	10000	3.10	3.48	7.15	7.33	149.73	154.61
20	500	0.61	1.14	0.02	0.02	0.56	0.78
	1000	0.51	0.87	0.07	0.07	2.15	2.63
	2500	0.88	1.40	0.42	0.45	13.01	13.68
	5000	1.58	1.95	1.68	1.70	53.45	54.99
	10000	2.96	3.74	6.69	6.71	346.58	353.68
30	500	1.62	4.73	0.02	0.03	0.76	0.89
	1000	0.87	1.95	0.06	0.07	3.32	3.63
	2500	0.99	1.25	0.45	0.46	19.58	20.20
	5000	1.62	2.59	1.66	1.68	79.76	83.42
	10000	4.82	8.07	6.58	6.74	526.61	549.03

Table 2.6 Comparison of the approaches on KPS benchmark instances (from [17]): Average and maximum time (s) for computing the optimal solutions.

For the instances considered, also our dynamic programming algorithm is capable of finding all optima with limited computational effort: it solves

all instances in at most 9 seconds (required for one of the largest instances involving 10000 items), but usually much less. The approach outperforms the dynamic programming algorithm in [17] and the differences in computational times increase as the size of the instances increases. We again remark that tests in [17] were performed on a slightly less performing machine. Anyhow, the significant differences, in particular on large instances, illustrate that the speed-up we obtain is much larger than the improvement attributed to the different hardware and underlines the impressive effectiveness of our approach. Notice also that the order of magnitude of costs/profits and setups/weights of the families/items is similar for the instances considered. Therefore the mere fact that our algorithm runs over profits rather than weights is not expected to be significant in explaining the differences of the performances. Moreover, our dynamic programming compares favorably also to $FLEA$ running on a similar machine. Computational performances are fully comparable and our dynamic programming can be regarded as a valuable alternative to the exact enumerative approach but without the necessity to use an ILP-solver.

It might be noted that for a direct comparison to [17] we rely on the strongly correlated instances available on-line. To get a broader picture we also ran tests on *weakly-correlated* instances, where profits were chosen as $p_{ij} = w_{ij} + u$ with u uniformly distributed in $[-10, 10]$. Comparing the performance of our dynamic programming algorithm with $FLEA$ the general trends observed above were confirmed. Both algorithms managed to solve these weakly-correlated instances in roughly half the time required for the strongly-correlated instances.
We finally remark that, in our computational experiments, we decided to stick to the generation schemes of the instances in the literature. We leave to future research the construction of instances involving other classical correlations between profits and weights of the items.

2.6 Approximation results

In this section we investigate the properties of KPS and conditions which allow the construction of fully polynomial time approximation schemes (FPTAS) for the problem. For the standard knapsack problem the performance analysis of

simple approximation algorithms, such as the greedy algorithm, often relies on a comparison with the linear relaxation of KP. The following consideration shows that this is not a meaningful strategy for KPS.

2.6.1 Linear relaxation KPS^{LP}

It was shown in [89] that the linear relaxation of KPS has a special structure and can be solved in linear time by a reduction to the linear relaxation of a standard KP as follows. Consider the items sorted in nonincreasing order of their efficiency p_{ij}/w_{ij}. Then for each family compute the *maximum reachable efficiency*, which is given by the first k_i^* items of family i with:

$$k_i^* = \arg\max_k \frac{\sum_{j=1}^{k} p_{ij} - f_i}{\sum_{j=1}^{k} w_{ij} + d_i} \quad (k = 1, \dots, n_i) \quad \forall i = 1, \dots, N \tag{2.26}$$

These items are grouped into a new virtual item for each family with profit and weight equal to $\sum_{j=1}^{k_i^*} p_{ij} - f_i$ and $\sum_{j=1}^{k_i^*} w_{ij} + d_i$ respectively. Then consider an instance of KP consisting of N virtual items and all the original items with capacity b. Its linear relaxation can be solved by applying Dantzig's rule and the optimal solution of this problem yields an optimal solution of the linear relaxation of KPS as well (see [89] for a complete proof).

In an optimal solution at most one variable is fractional, either an original item or a virtual item. In the former case an item $x_{\ell j}$ is fractional while the corresponding family ℓ is selected together with all its items of efficiency greater than $p_{\ell j}/w_{\ell j}$. In the latter case a fractional part of all the most effective items, namely the items x_{hj} with $j \leq k_h^*$, of a specific family h is considered. This reformulation of the problem allows us to show that the optimal solution value of the LP-relaxation can exceed the optimal solution value of KPS by an arbitrarily large quantity.

Proposition 1. *The difference $z^{LP} - z^*$ can be arbitrarily large.*

Proof. Consider the following KPS instance with $N = 1$ and capacity $b = M + 1$ with M being an arbitrary large integer number: There is $f_1 = M - 1$, $d_1 = M$ and $n_1 = 2$ identical items with $p_{1j} = M$ and $w_{1j} = 1$.

The optimal solution of this problem can pack only one item with total profit $p_{1j} - f_1 = 1$. The linear relaxation distributes the setup weight equally over the two items and sets $y_1 = x_{1j} = \frac{M+1}{M+2}$ which gives a total weight of

$$2\frac{M+1}{M+2} + M\frac{M+1}{M+2} = M + 1 = b.$$

This yields a total profit of

$$2M\frac{M+1}{M+2} - (M-1)\frac{M+1}{M+2} = (M+1)\frac{M+1}{M+2} \approx M,$$

which can become arbitrarily large compared to the integer optimal solution value of 1. □

Thus, for KPS we cannot derive an algorithm with bounded performance guarantee by exploiting the properties of the linear relaxation, as it was done for the standard KP.

2.6.2 Negative approximation result

We prove here the more general result that no polynomial time approximation algorithm exists for KPS (under $\mathcal{P} \neq \mathcal{NP}$).

Theorem 2. *KPS does not have a polynomial time approximation algorithm with a bounded approximation ratio unless $\mathcal{P} = \mathcal{NP}$.*

Proof. The theorem is proved by reduction from the Subset Sum Problem (SSP) cited in Section 1.7, where n items j with integer weights w'_j (with $j = 1, \ldots, n$) and a value W' (with $\sum_{j=1}^{n} w'_j > W'$) are given. The decision version of SSP is a well–known NP–complete problem and asks whether there exists a subset of items represented by x^* such that $\sum_{j=1}^{n} w'_j x^*_j = W'$.

We build an instance of KPS with just one family with setup cost and weight $f_1 = d_1 = W' - 1$, profits and weights of the $n_1 = n$ items $p_{1j} = w_{1j} = w'_j$

(with $j = 1, \ldots, n$) and capacity $b = 2W' - 1$. The capacity bound implies that for every feasible solution x_{1j} there is $\sum_{j=1}^{n_1} p_{1j}x_{1j} = \sum_{j=1}^{n_1} w_{1j}x_{1j} \leq b - d_1 = W'$. Hence, the optimal solution of this KPS instance is bounded by $\sum_{j=1}^{n_1} p_{1j}x_{1j} - (W' - 1) \leq 1$. Not activating the family at all yields the trivial solution with profit 0. By integrality of the input data this limits the optimal solution value to 0 or 1, where the latter value can be attained if and only if SSP has a solution.

Thus if there was a polynomial time approximation algorithm with a bounded approximation ratio, we could resolve SSP just by checking if the approximate solution of KPS is strictly positive. Clearly this is not possible under the assumption that $\mathcal{P} \neq \mathcal{NP}$. □

Note that in the KPS-instance of the above proof all items have identical profits and weights and this applies also to the setup values. Thus, the result of Theorem 2 holds also for the "Subset Sum" variant of KPS.

2.6.3 Four special cases of KPS

Given this general inapproximability result, it is interesting to investigate to what extent approximation algorithms can be derived when instances with restricting assumptions are considered. In fact, we try to characterize the border between non-approximability and existence of approximation schemes by analyzing four relevant special cases of KPS.

Case 1, each family can be packed: $d_i + \sum_{j=1}^{n_i} w_{ij} \leq b \quad \forall i = 1, \ldots, N$

This case considers the situation where each single family can be packed into the knapsack with all its items. We first show that a constant performance guarantee exists under this assumption. Consider the linear relaxation of KPS throughout the reduction to the classical knapsack problem as in Section 2.6.1. As in KP, we can consider a feasible solution of KPS consisting of those variables set to 1 in the optimal solution of the linear relaxation. Denote by z^{LG} the value of such a solution. Then we introduce the following *KPS–Greedy*

algorithm with objective function value z^G.

$$z^G = \max\{z^{LG}, p_{max} = \max_i \left(\sum_{j=1}^{n_i} p_{ij} - f_i \right)\} \tag{2.27}$$

That is, the *KPS–Greedy* algorithm simply takes the best solution among the one given by the variables set to 1 in the optimal solution of the linear relaxation and the ones provided by packing one family only. In addition, it is straightforward to see that for the optimal solution value z^* of KPS the following inequality holds, regardless if the fractional variable of the linear relaxation is an existing item or a virtual item:

$$z^* < z^{LG} + p_{max} \tag{2.28}$$

Consequently we can easily show that *KPS–Greedy* is a 1/2–approximation algorithm for KPS.

Proposition 2. *Algorithm KPS–Greedy has a relative performance guarantee of* 1/2 *and this bound is tight.*

Proof. Consider the approximation ratio $\rho = \frac{\max\{z^{LG}, p_{max}\}}{z^*}$ for an arbitrary instance of KPS. We get with (2.28):

$$\rho > \frac{\max\{z^{LG}, p_{max}\}}{z^{LG} + p_{max}} \geq \frac{\max\{z^{LG}, p_{max}\}}{2\max\{z^{LG}, p_{max}\}} = \frac{1}{2}$$

We can show that the ratio $\frac{1}{2}$ is tight by considering the following example with $N = 2$ families each with $n_i = 2$ and the following entries:

$$b = 2(M+1);$$
$$f_1 = 1,\ d_1 = 1,\ p_{11} = 1,\ p_{12} = M,\ w_{11} = 1,\ w_{12} = M;$$
$$f_2 = 2,\ d_2 = M,\ p_{21} = M,\ p_{22} = 1,\ w_{21} = 1,\ w_{22} = M;$$

M is an arbitrary large integer value. The *maximum reachable efficiency* of the families is equal to $\frac{M}{M+2}$ for family 1 and $\frac{M-2}{M+1}$ for family 2. Since the first family has a greater efficiency ($\frac{M}{M+2} - \frac{M-2}{M+1} = \frac{M+4}{(M+2)(M+1)} > 0$), the linear relaxation sets $y_1 = x_{11} = x_{12} = 1$ and $y_2 = x_{21} = \frac{M}{M+1}, x_{22} = 0$. Therefore

algorithm *KPS–Greedy* considers a solution where only the items of the first family are packed into the knapsack with an approximate solution value $z^G = M$ (since $z^{LG} = p_{max} = M$).

The optimal solution instead involves the selection of the second item of the first family and of the first item of the second family with optimal solution value equal to $2M - 3$, which implies that the approximation ratio can be arbitrarily close to $\frac{1}{2}$ as the value of M increases. □

For the case at hand an FPTAS can also be derived.

Theorem 3. *There is an FPTAS for KPS if each family can be packed into the knapsack.*

Proof. First we apply the well–known scaling technique of the profits adopted for the standard knapsack problem. More precisely, we round down the profits of the items p_{ij} and round up the setup costs of the families f_i by considering a scaling factor K yielding the following scaled values:

$$\tilde{p}_{ij} = \left\lfloor \frac{p_{ij}}{K} \right\rfloor \quad \forall\ i,j \text{ and } \tilde{f}_i = \left\lceil \frac{f_i}{K} \right\rceil \quad \forall\ i \tag{2.29}$$

Running the dynamic programming depicted in Section 2.4.2 on the problem with scaled profits and costs we get an optimal solution denoted by the sets of the items and families $\tilde{X}$ and $\tilde{Y}$ respectively. We also indicate by z^A the approximate solution value of KPS with items and families in $\tilde{X}$ and $\tilde{Y}$. Finally we denote an optimal solution of KPS by X^* and Y^* (with value z^*). The following series of inequalities holds:

$$
\begin{aligned}
z^A &= \sum_{ij \in \tilde{X}} p_{ij} - \sum_{i \in \tilde{Y}} f_i \\
&\geq \sum_{ij \in \tilde{X}} K \left\lfloor \frac{p_{ij}}{K} \right\rfloor - \sum_{i \in \tilde{Y}} K \left\lceil \frac{f_i}{K} \right\rceil \\
&\geq \sum_{ij \in X^*} K \left\lfloor \frac{p_{ij}}{K} \right\rfloor - \sum_{i \in Y^*} K \left\lceil \frac{f_i}{K} \right\rceil \\
&\geq \sum_{ij \in X^*} K \left(\frac{p_{ij}}{K} - 1 \right) - \sum_{i \in Y^*} K \left(\frac{f_i}{K} + 1 \right) = \\
&= z^* - K \left(|X^*| + |Y^*| \right)
\end{aligned}
\tag{2.30}
$$

Therefore the dynamic programming has a performance guarantee of $(1-\varepsilon)$ if:

$$
K \leq \frac{\varepsilon z^*}{|X^*| + |Y^*|} \tag{2.31}
$$

Considering that the cardinality of both X^* and Y^* is trivially bounded by the total number of items n, we can easily satisfy condition (2.31) by setting $K = \frac{\varepsilon p_{max}}{2n}$.

The running time is dominated by executing the dynamic program on the scaled items. A simple upper bound on the profit range of the problem with scaled items is given by $n \frac{p_{max}}{K} = 2n^2 \frac{1}{\varepsilon}$. This yields an overall running time of $O(n^3 \frac{1}{\varepsilon})$, which establishes an FPTAS for KPS. □

Case 2, bounded setup costs: $f_i < \max(p_{ij}) \quad j = 1, \ldots, n_i \quad \forall i = 1, \ldots, N$

We consider now the case in which the setup costs of the families are strictly bounded by the value of their most profitable items. As in the general case, we prove that there is no approximation algorithm unless $\mathcal{P} = \mathcal{NP}$.

Theorem 4. *For KPS with $f_i < \max(p_{ij})$ no polynomial time approximation algorithm with bounded approximation ratio exists unless $\mathcal{P} = \mathcal{NP}$.*

Proof. The theorem is proved again by reduction from the Subset Sum Problem involving n items x_j with integer weights w'_j (with $j = 1, \ldots, n$) and subset sum W' and employs a refinement of the proof of Theorem 2. Let us assume

that an approximation ratio $\rho > 0$ (possibly as a function $\rho(n)$) exists. We build an instance of KPS with one family with weight $d_1 = 0$, setup cost $f_1 = (\frac{1+\rho}{\rho})(W' - 1)$ and capacity $b = W'$. Consider then $n+1$ items with weights $w_{1j} = w'_j$, profits $p_{1j} = (\frac{1+\rho}{\rho})w'_j$ for $j = 1, \ldots, n$ and $w_{1(n+1)} = W'$, $p_{1(n+1)} = f_1 + 1$. The optimal solution value of this KPS instance can be:

- $z^* = 1$ if SSP has answer "No". In this case the best option is to place into the knapsack only the item $(n+1)$ reaching a profit equal to 1, since the other items cannot produce a whole positive profit: $(\frac{1+\rho}{\rho})(\sum_{j=1}^{n} w_{1j}x_{1j} - W' + 1) \leq 0$.
- $z^* = \frac{1+\rho}{\rho} > 1$ if SSP has answer "Yes". In such a case including into the knapsack the items corresponding to a feasible solution of SSP dominates the alternative of packing only the item $(n+1)$.

Therefore, if SSP admits a solution, a ρ–approximation algorithm would yield a solution value for KPS $z^A \geq \rho(\frac{1+\rho}{\rho}) > 1$. Similarly to the general case, simply checking that z^A is strictly greater than 1 would decide a NP–complete problem contradicting the assumption that $\mathcal{P} \neq \mathcal{NP}$. □

Case 3, bounded setup costs: For some constant $k \geq 1$ there is $f_i \leq k \cdot \min(p_{ij}) \quad j = 1, \ldots, n_i \quad \forall i = 1, \ldots, N$

This case refers to the situation where the setup costs of each family do not exceed the lowest profit of their items by more than a constant factor. In particular the sub–case where k=1 represents the plausible assumption that every item gives a positive contribution, even on its own. First we compute suitable lower and upper bounds for the optimal solution of KPS. Then we rely on the scaling technique discussed in Case 1 in order to derive an FPTAS.

For each family, we consider all ℓ-tuples of items as possible solutions for $1 \leq \ell \leq k$, which takes $O(n^k)$ time. Moreover, in $O(n^k)$ time we can also compute a $\frac{k+1}{k+2}$–approximation algorithm for the standard knapsack problem induced by selecting family i with capacity equal to $b - d_i$, see [47, Sec. 6.1].

Denote by z_i^{AKP} the approximate solution value of the classical knapsack and by z_i^{AKPS} the corresponding KPS value: $z_i^{AKPS} = z_i^{AKP} - f_i$.

Likewise, we indicate the optimal value of the knapsack problem induced by family i by z_i^{OKP} and the corresponding KPS solution value by $z_i^{OKPS} = z_i^{OKP} - f_i$. We can show:

Proposition 3. $z_i^{AKPS} \geq \frac{z_i^{OKPS}}{(k+1)(k+2)} \quad \forall i = 1, \ldots, N$

Proof. We prove our claim by distinguishing if the optimal solution of the standard knapsack consists of at most k or more than k items. In the former case the approximation algorithm for the standard knapsack problem would find the optimal solution and thus $z_i^{AKP} = z_i^{OKP}$ and $z_i^{AKPS} = z_i^{OKPS}$. In the latter case at least $k+1$ items are included in the optimal solution implying that $z_i^{OKP} \geq (k+1)\min_j p_{ij} \geq (k+1)\frac{f_i}{k}$. Then the following inequalities hold:

$$\begin{aligned} z_i^{AKPS} &= z_i^{AKP} - f_i \\ &\geq z_i^{AKP} - \frac{k}{k+1} z_i^{OKP} \\ &\geq \frac{k+1}{k+2} z_i^{OKP} - \frac{k}{k+1} z_i^{OKP} \\ &= \frac{k^2+2k+1-k^2-2k}{(k+1)(k+2)} z_i^{OKP} = \frac{z_i^{OKP}}{(k+1)(k+2)} \\ &\geq \frac{z_i^{OKPS}}{(k+1)(k+2)} \end{aligned}$$

which completes the proof. □

If we take the maximum among the z_i^{AKPS} and denote this value by z^{AKPS}, we have with Proposition 3:

$$z^{AKPS} = \max_i (z_i^{AKPS}) \geq \max_i \left(\frac{z_i^{OKPS}}{(k+1)(k+2)} \right) \geq \frac{z^*}{N(k+1)(k+2)} \tag{2.32}$$

Therefore, we obtain upper and lower bounds on the optimal value z^* of KPS:

$$z^{AKPS} \leq z^* \leq N(k+1)(k+2) z^{AKPS} \tag{2.33}$$

We remark that other procedures to further narrow the interval around z^* may be devised, but this is out of the scope of the present work. Anyhow given the lower and upper bounds in (2.33) we can prove that:

Theorem 5. *There is an FPTAS for KPS with* $f_i \leq k \cdot \min(p_{ij})$ *for any positive constant* k.

Proof. The theorem is proved by simply applying the scaling technique as in Case 1 and by setting $K = \frac{\varepsilon z^{AKPS}}{2n}$, a value that satisfies condition (2.31). An upper bound on the profit range for the problem with scaled items and families follows from (2.33), namely $N(k+1)(k+2)z^{AKPS}\frac{1}{K} = 2nN(k+1)(k+2)\frac{1}{\varepsilon}$. Also in this case the dynamic programming algorithm implies an FPTAS with running time complexity $O(n^3\frac{1}{\varepsilon} + n^k)$. □

Case 4, families of bounded size: $n_i \leq C \quad \forall i = 1, \ldots, N$

We analyze here the case where the number of items in each family is bounded by a constant value C. Under this assumption, it is convenient to see KPS as a special case of the Multiple Choice Knapsack Problem (MCKP) introduced in Section 1.7. We outline the reduction of KPS to MCKP in the following. Associate with a class of MCKP a family from KPS. Then for each possible subset of items in a given family create an item in the corresponding class of MCKP with:

- a profit equal to the sum of the profits of the subset minus the setup cost of their family;
- a weight equal to the sum of the weights of the subset plus the setup weight of the corresponding family.

An item with profit and weight equal to 0 corresponds to the situation in which the family is not selected in a solution of KPS. Subsets yielding a non–positive profit are discarded. Finally, setting the capacity of MCKP equal to b completes the reduction. Since each class of MCKP has a cardinality bounded by the constant 2^C, the cardinality of the power set of the largest family, the total number of items in MCKP is bounded by $N2^C$. It follows immediately from the corresponding results for MCKP that there is a $1/2-$approximation algorithm for this special case of KPS with running time $O(N2^C)$ and also a $4/5-$approximation in $O(N \log N2^C)$ time. If we consider the FPTAS for

MCKP as outlined in [47, ch. 11] with complexity $O(\frac{n'm'}{\varepsilon})$, where n' and m' indicate the number of items and classes respectively, we get an FPTAS for KPS as well with running time complexity $O(\frac{N^2}{\varepsilon}2^C)$. Thus we can state the following theorem:

Theorem 6. *KPS with $n_i \leq C$ for a constant C admits an FPTAS.*

The 0–1 Collapsing Knapsack Problem

3.1 Introduction

The 0–1 Collapsing Knapsack Problem (CKP) can be seen as a generalization of the standard 0–1 Knapsack Problem (KP) where the capacity of the constraint is not a scalar but a non–increasing function of the number of included items, namely, it is inversely related to the number of items placed inside the knapsack. While KP has been widely studied, CKP has gained less attention. According to [82], CKP has wide applications such as in satellite communication and time–sharing computer systems, namely in problems where a structural overhead is induced by the number of items or users considered. In a satellite communication, a correct transmission on the band requires that the parts of the band dedicated to each user must be separated by proper gaps. In time–sharing computer systems, just the adding of a process while other processes are running causes an overhead of the processing capabilities. Another application of interest is the transportation of fragile items, which may require additional coverings if they are transported with other items. These and similar real-life applications can be modeled as Collapsing Knapsack Problems where the non–increasing function of the capacity represents the overhead of the resources produced by the number of items included in a solution.

CKP was first introduced in [82], where an implicit enumeration algorithm was proposed. An exact algorithm making use of new upper and lower bounds was presented in [30]. In [71], a pseudopolynomial time dynamic programming approach was proposed together with a reduction scheme to the standard knapsack problem. An improved reduction scheme is proposed in [42]. Finally,

an exact algorithm based on the partitioning of the CKP into sub–problems was presented in [88].

The contribution of our work is twofold. On the one hand, we present a novel ILP formulation of CKP and an effective reduction procedure for restricting the solution space of the problem. We remark that our novel ILP formulation, despite its simplicity, provides a significant contribution for tackling the CKP since it makes possible to exploit the potentials of the modern IP solvers. The previous formulation of the problem is not linear and the reduction schemes to a standard KP illustrated in [71] and [42] induce very large size coefficients that make the KP very difficult to solve in practice.
On the other hand, we introduce an exact approach for CKP which is also extended to the multidimensional variant of CKP denoted hereafter M-CKP (with $M > 1$ indicating the number of capacity constraints). To the best of the author's knowledge, no work has been developed to tackle collapsing knapsack problems with more than one capacity constraint. We propose a new exact approach that relies on the ILP formulation of CKP and on an original branching scheme that induces the solution of several KPs (with the additional constraint that the number of items in the knapsack is fixed) by exploiting the particular structure of CKP. In this respect, the approach shares the general algorithmic idea proposed for KPS of inducing problems tractable in practice through an effective exploration of the solution space of first level variables. The proposed approach is capable of solving to optimality all instances with up to 100000 items within a time limit of 600 seconds, while instances tackled in the literature until now were limited to 1000 items in size. The exact approach is capable of solving to optimality also 2-CKP in instances up to 100000 items in reasonable time. For M-CKP with M $= 3, 4, 5$, the proposed approach is capable of solving to optimality all instances with up to $16000, 1500$ and 1000 respectively within a time limit of 3600 seconds.

These contributions resulted in a paper published in a journal ([26]). I presented the results on CKP at EURO 2015 conference and as finalist of the Prix Jeune Chercheur at ROADEF 2015 conference.
The chapter is organized as follows. In Section 3.2, the new integer linear programming formulation of the problem is provided together with its general-

ization to the multidimensional CKP. In Section 3.3 the reduction procedure is introduced. Section 3.4 is devoted to the description of the proposed exact solution algorithm. In Section 3.5, computational results for CKP and M-CKP are presented.

3.2 ILP modeling of CKP and M-CKP

In this section, we briefly recall the original formulation of CKP and we investigate the effectiveness of two schemes laid out in the literature for reducing CKP to the classical knapsack problem. After that, we present our novel ILP formulation and its extension to the multidimensional variant of the problem.

3.2.1 A previous formulation of CKP

According to [88], the 0–1 Collapsing Knapsack Problem (CKP) can be expressed as follows:

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i \tag{3.1}$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq B\left(\sum_{i=1}^{n} x_i\right) \tag{3.2}$$

$$x_i \in \{0,1\} \qquad i = 1,\ldots,n \tag{3.3}$$

where p_i and w_i, positive integers, denote *profit* and *weight* of each item i. Function $B(\cdot)$ is non-increasing over $\{1,2,\ldots,n\}$, indicating the capacity of the knapsack. This implies that the capacity will decrease while the number of the items inserted into the knapsack increases. Each binary variable x_i indicates if item i is selected or not. CKP is known to be weakly NP–hard as it is pseudopolynomially solvable [71] and contains KP as special case (when function $B(\cdot)$ is a constant).

Naively, according to model (3.1)–(3.3), one can enumerate all possible sub–problems by iteratively fixing the total number of items and taking the

corresponding capacity value from the function $B(\cdot)$. Then, one can solve all the sub–problems and consider the best solution among them. This task corresponds for each sub–problem to solve a KP problem with the additional constraint that the number of items in the knapsack is fixed, which is generally handled well by simply using a standard ILP solver. However, this approach is not expected to be effective as soon as the number of variables, and correspondingly the number of possible capacity values, increases.

3.2.2 Reduction schemes of CKP to a standard KP

In [71] a reduction scheme of CKP to a pure KP is proposed. That reduction relies on doubling the number of variables and introducing coefficients of very large size. The authors of [71], however, indicated that the presence of very large coefficients made those KP intractable in practice. Indeed, in [57], *Combo*, the current state-of-the-art algorithm for KP, was limited to instances with up to 200 items only, as the generation of larger instances was not possible on the machine used by the authors. Preliminary computational tests on that reformulation confirmed this fact. We considered CKP instances with 1000 items generated as the largest instances in [71]. Then, we launched CPLEX 12.5 for solving the standard knapsack problem produced by the reduction scheme: a CPU time limit of 1200 seconds was not sufficient to reach the optimal solution. An improved reduction scheme is proposed in [42]. The scheme produces coefficients smaller than those in [71]. We tested this scheme by CPLEX 12.5 as well. Such a scheme was able to solve to optimality the 1000-item instances in few seconds but was not able to reach the optimum on instances with 2000 items, within a CPU time limit of 1200 seconds.

In the light of these considerations, the reduction schemes in the literature are not appealing in practice. We propose instead, in the following, a new ILP formulation of CKP and M-CKP that constitutes the basic element of the proposed solution approach.

3.2.3 A novel ILP formulation

It is possible to formulate explicitly function $B(\cdot)$ in CKP, by adding a new set of 0–1 variables and two constraints in order to deal with the non–increasing property of function $B(\cdot)$. Let denote by $b_j = B(j)$ $(j = 1,\dots,h)$ the h (with $h \leq n$) possible *capacity values* associated with the sum of selected items x_i. We introduce h 0–1 variables y_j indicating whether a specific *capacity value* j is selected or not. Then, we have the following ILP model for CKP:

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i \tag{3.4}$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq \sum_{j=1}^{h} b_j y_j \tag{3.5}$$

$$\sum_{j=1}^{h} y_j = 1 \tag{3.6}$$

$$\sum_{i=1}^{n} x_i = \sum_{j=1}^{h} j y_j \tag{3.7}$$

$$x_i \in \{0,1\} \qquad i = 1,\dots,n \tag{3.8}$$

$$y_j \in \{0,1\} \qquad j = 1,\dots,h \tag{3.9}$$

Constraint (3.5) links the weighted sum of the items to all possible capacity values. Constraint (3.6) ensures that exactly one *capacity value* is selected. Constraint (3.7) relates an index j to the total number of items selected in the solution. In other words if a variable y_j is selected, then we guarantee that the capacity value b_j corresponds to the associated number of selected items. Finally in (3.8) and (3.9) variables x_i and y_j are defined as binary.

We remark that a simple upper bound on the total number of the items present in the optimal solution can be straightforwardly computed. Let denote by means of square brackets the set of weights sorted in nondecreasing order so that $w_{[1]} \leq w_{[2]} \leq \dots \leq w_{[n]}$. The following Property which is a reformulation of Proposition 5 in [71] holds.

Property 1. *(Proposition 5 in [71]) Let* $k^* = \min\{k | \sum_{j=1}^{k} w_{[j]} > b_k\}$. *Then, without loss of optimality,* $y_l = 0 \ \forall l \in [k^*, ..., h]$.

From Property 1, the following constraint is added without loss of generality to (namely the relevant variables are deleted from) model (3.4)–(3.9).

$$y_l = 0 \ \forall l \in [k^*, ..., h] \tag{3.10}$$

From now on, we will denote model (3.4)–(3.10) as model CKP_a.

This novel ILP model turns out to be quite effective in tackling large size instances of the problem, just by allowing the use of a MIP solver like CPLEX 12.5. More precisely, CPLEX 12.5 manages to solve CKP instances with up 30000 items within a CPU time limit of 600 seconds. This constitutes already a significant improvement of the results reached by the previous approaches in the literature.

3.2.4 Extending the ILP formulation to M-CKP

M-CKP can be seen as a generalization of the multidimensional knapsack problem, that is it contains the multidimensional knapsack problem as a special case when the functions associated to the capacities are constants. Model (3.4)–(3.9) can be extended to M-CKP by replacing constraint (3.5) with the following set of constraints:

$$\sum_{i=1}^{n} w_i^{(m)} x_i \leq \sum_{j=1}^{h} b_j^{(m)} y_j \quad m = 1, \ldots, M \tag{3.11}$$

Where $w_i^{(m)}$ and $b_j^{(m)}$ denote the weights of the items and the capacities values in the capacity constraint m. Similarly to CKP_a, an upper bound on the number of items for M-CKP can be computed by applying Property 1 to all capacity constraints. Let denote by $k_1^*, ..., k_M^*$ the max number of items in

each of the capacity constraints. Let us consider the minimum of these values denoted by $t = \min(k_1^*, ..., k_M^*)$. Then, the following constraint is added without loss of generality to the model:

$$y_l = 0 \ \ \forall l \in [t, ..., h] \tag{3.12}$$

Henceforth we will indicate model (3.4), (3.6)–(3.9), (3.11)–(3.12) as model M-CKP_a.

In the following, we analyze structure and properties of model CKP_a. We first devise a simple reduction procedure that shows up to effectively restrict the solution space and improve the performance of CPLEX 12.5. Then, we propose an exact solution approach that takes in input the output of the reduction procedure and related reduced ILP model and provides in output the optimal solution. The approach is outlined for CKP (the extension to M-CKP being generally straightforward) with few integrations for M-CKP when required.

3.3 A reduction procedure for CKP

We propose a reduction procedure that consists in finding a first solution and correspondingly reducing the relevant sums of items that may lead to an optimal solution. A further step seeks for variables to be univocally fixed to 0 or 1 without loss of optimality. Each step of the procedure is detailed in the following.

3.3.1 Computing a first solution

The goal of this step is to effectively compute a first solution for CKP strongly based on the optimal solution of the related continuous relaxation.

First, we consider and solve the continuous relaxation CKP_a^{LP} of model CKP_a. Flooring the optimal sum of items of this problem provides a possible value S of the total number of items in the optimal solution of CKP_a : $S' = \lfloor \sum_{i=1}^{n} x_i^{LP} \rfloor$. We remark that S' always constitutes a feasible sum of items for

CKP_a, given the initial computation of the upper bound on the number of items obtained by means of Property 1. Then, we consider again model CKP_a where, however, we add the constraint $y_{S'} = 1$, that is: $\sum_{i=1}^{n} x_i = S'$. This corresponds to solve the following model (denoted by $CKP_a(S')$).

$CKP_a(S')$:

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i \tag{3.13}$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq b_{S'} \tag{3.14}$$

$$\sum_{i=1}^{n} x_i = S' \tag{3.15}$$

$$x_i \in \{0,1\} \qquad i = 1,\ldots,n \tag{3.16}$$

Model $CKP_a(S')$ is a KP with the additional constraint that the number of items is given (for a discussion of knapsack problems with cardinality constraints see [12]). We first consider its continuous relaxation $CKP_a(S')^{LP}$.

The following Lemma holds.

Lemma 1. *The optimal solution of $CKP_a(S')^{LP}$ has either zero or two fractional variables.*

Proof. It is well known from standard textbooks of linear programming (see, for instance, [68]) that for any continuous LP model with p constraints and bounded variables, there exists at least one optimal basic solution with p basic variables where only the basic variables can have a value that differs from the bounds. Hence, in this case, with 2 constraints at most 2 variables can have fractional value. Besides, the optimal solution of $CKP_a(S')^{LP}$ cannot have just one fractional variable or else $\sum_{i=1}^{n} x_i$ would not be integer violating constraint (3.15) as S' is integer. □

If the optimal solution of $CKP_a(S')^{LP}$ has zero fractional variables, it constitutes an optimal integer solution of $CKP_a(S')$ as well. Alternatively, there are two fractional variables x_i, x_j and a feasible solution for $CKP_a(S')$ can be obtained by taking between i and j the item with the lowest weight

and discarding the other one (see Lemma 1 in [12]). Denote the value of this heuristic solution as LB'.

For M-CKP, the following Corollary holds.

Corollary 1. *The optimal solution of M-$CKP_a(S')^{LP}$ has either zero or from 2 to $M+1$ fractional variables.*

Proof. By applying the same reasoning of Lemma 1, the result holds. □

In this latter case, a feasible integer solution can be found by trying to insert in the solution either one or more of the fractional items. This is done by solving model M-$CKP_a(S')$ modified by adding constraints

$$x_i = x_i^{LP}(S') \quad \forall i : x_i^{LP}(S') \text{ is integer} \tag{3.17}$$

and by replacing equation (3.15) with inequality

$$\sum_{i=1}^{n} x_i \leq S' \tag{3.18}$$

where $x_i^{LP}(S')$ denote the optimal values of variables x_i in M-$CKP_a(S')^{LP}$.

Due to Corollary 1, the additional constraints induce a resulting model with at most $M+1$ variables that is therefore immediate to solve in practice when M is limited by means of a MIP solver like CPLEX 12.5. Notice also that, even if none of the fractional variables can be selected, we still get a feasible solution for M-CKP_a by simply keeping the rest of the solution, as the value of the sum of the items in this case is less than S'.

3.3.2 Limiting the relevant range of sums of items

After the computation of the first solution LB', the possible sums of the items in an optimal solution can be straightforwardly limited by solving other two

continuous problems. More precisely, we minimize and maximize the sum of items $\sum x_i$ subject to constraints (3.5)–(3.7), (3.10) and to an additional constraint ensuring that the total profit must be strictly greater than the current solution value. Notice also that the sum S' of the items in the first solution provides an upper bound and a lower bound for the minimization and maximization problem respectively. Thus further constraints on the y_i variables can be added. The corresponding models to solve (denoted by CKP_{min} and CKP_{max} respectively) are as follows.

CKP_{min}:

$$\min \quad \sum_{i=1}^{n} x_i \tag{3.19}$$

$$\text{subject to} \quad \sum_{i=1}^{n} p_i x_i \geq LB' + 1 \tag{3.20}$$

$$\sum_{j=S'}^{h} y_j = 0 \tag{3.21}$$

$$(3.5) - (3.7), (3.10)$$

$$0 \leq x_i \leq 1 \qquad i = 1, \ldots, n \tag{3.22}$$

$$0 \leq y_j \leq 1 \qquad j = 1, \ldots, h \tag{3.23}$$

CKP_{max}:

$$\max \quad \sum_{i=1}^{n} x_i \tag{3.24}$$

$$\text{subject to} \quad \sum_{i=1}^{n} p_i x_i \geq LB' + 1 \tag{3.25}$$

$$\sum_{j=1}^{S'} y_j = 0 \tag{3.26}$$

$$(3.5) - (3.7), (3.10)$$

$$0 \leq x_i \leq 1 \qquad i = 1, \ldots, n \tag{3.27}$$

$$0 \leq y_j \leq 1 \qquad j = 1, \ldots, h \tag{3.28}$$

Ceiling and flooring the optimal solution values of CKP_{min} and CKP_{max} yield the extremes of the range of the number of items possibly leading to

an optimal solution of CKP. Let us denote these extremes by S_{min} and S_{max}. At the end of this step, the following constraints are added without loss of optimality:

$$y_j = 0 \ \forall j \in [1, ..., S_{min} - 1] \tag{3.29}$$

$$y_j = 0 \ \forall j \in [S_{max} + 1, ..., h] \tag{3.30}$$

3.3.3 Variables fixing

We get back then to model CKP_a^{LP}. Let indicate the optimal value of CKP_a^{LP} by z^{LP} and the optimal values of variables x_i and y_i by x_i^{LP} and y_i^{LP}, respectively. Let r_{x_i} and r_{y_i} be the reduced costs of non basic variables in the optimal solution of CKP_a^{LP}. We apply then the standard variable-fixing techniques from Integer Linear Programming as shown for KPS in Section 2.3.2. Thus, the following constraints are added to the models.

$$\forall i : |r_{x_i}| \geq z^{LP} - LB', \quad x_i = x_i^{LP} \tag{3.31}$$

$$\forall j : |r_{y_j}| \geq z^{LP} - LB', \quad y_j = y_j^{LP} \tag{3.32}$$

This step is fast, since it relies on solving continuous problems with a limited number of constraints. At the same time, it shows up to be very effective in reducing the problem size and in our case (in general, fixing variables may sometimes generate instances that are more difficult to solve than the original ones) also in strongly improving the performance as indicated in the computational results section. The pseudo code of the reduction procedure is outlined in Algorithm 3.

Algorithm 3 Reduction procedure

Input: CKP instance.
▷ Finding an initial feasible solution
$\boldsymbol{x}^{LP} \leftarrow$ solve model CKP_a^{LP};
$S' \leftarrow \lfloor \sum_{i=1}^{n} x_i^{LP} \rfloor$;
$\boldsymbol{LB'} \leftarrow$ solve model $CKP_a(S')^{LP}$ and apply Lemma 1;
▷ Identifying the relevant sums of items
$z_{min} \leftarrow$ solve model CKP_{min};
$z_{max} \leftarrow$ solve model CKP_{max};
$S_{min} = \lceil z_{min} \rceil$;
$S_{max} = \lfloor z_{max} \rfloor$;
Set in model CKP_a: $y_j = 0 \ \forall j \in [1, ..., S_{min} - 1]$ and
$y_j = 0 \ \forall j \in [S_{max} + 1, ..., h]$;
▷ Variables fixing
Apply (3.31, 3.32) and fix variables in model CKP_a;
return $\boldsymbol{LB'}$;

3.4 An exact solution approach

The proposed approach applies initially the reduction procedure that provides an initial feasible solution LB' and an ILP model with reduced size where, in particular, some of the y_j variables have typically already been fixed to 0 stating correspondingly that there exists at least one optimal solution with $\sum x_j \neq j$. Then, our exact approach aims to further reduce, with limited computational effort, the subset of variables y_j that may possibly lead to an optimal solution when set to 1, that is the relevant range of optimal sums over items. Consequently, the optimal solution for each element of this subset (where the knapsack capacity is given) is computed. The approach is based on three main steps. The first step consists in improving the initial feasible solution provided by the reduction procedure. The second step concerns the identification of the smallest subset of variables y_j that can lead to an optimal solution, that is the relevant range of optimal sums over items. In the third step, the corresponding sub-problems with given y_j are solved. In practice, we reconsider here the algorithmic idea proposed for the Knapsack Problem

with Setups and based on an effective exploration of the solution space of first level variables (y_j) in order to induce sub–problems tractable in practice even though NP–Hard (namely, in this case, KPs with a cardinality constraint).

3.4.1 Finding an improved starting feasible solution

Consider model CKP_a and relax the integrality constraint on the x_i variables only, that is substitute constraints (3.8) with constraints

$$0 \leq x_i \leq 1 \qquad i = 1, \ldots, n \tag{3.33}$$

so as to get a mixed integer model denoted here after model CKP_b where the y_j variables remain binary. We remark that the set of y_j variables represents a special ordered set and therefore it is likely that model CKP_b may constitute a problem relatively easy to solve. This consideration showed up to hold in practice. The optimal solution value of model CKP_b, provides, on the one hand, an upper bound on the optimum. On the other hand, it provides also a possible integer value S of the total number of selected items, where S is the number of selected items in the optimal solution of model CKP_b.

Consider then $CKP_a(S)$ that is another problem that shows up to be easy to solve in practice. Its solution provides a feasible solution for model CKP_a. Let denote by LB the value of the improved starting feasible solution, that is the maximum between LB' and the optimal solution value of $CKP_a(S)$.

3.4.2 Identifying the relevant sub–problems

Given a feasible solution with value LB with exactly S items included, we propose a binary branching scheme on the number of items present in the solution. At the first level of the search tree, then, and additional constraint is added in such a way that either $\sum_{i=1}^{n} x_i \leq S-1$ (left branch) or $\sum_{i=1}^{n} x_i \geq S+1$ (right branch). In the left branch, we solve model CKP_b subject to the above men-

tioned additional constraint and stop the analysis if the obtained upper bound is not superior to the current best available feasible solution. Alternatively, we store the information that $\sum_{i=1}^{n} x_i = S-1$ is a potential candidate for the optimal solution of model CKP_a and consider the case with $\sum_{i=1}^{n} x_i \leq S-2$. We continue then in this way until the stopping condition is met. Similar analysis holds on the right branch where first we consider $\sum_{i=1}^{n} x_i \geq S+1$, then $\sum_{i=1}^{n} x_i \geq S+2$ and so on. At the end of this step, we have covered the relevant range on the number of items that may be present in the optimal solution of model CKP_a. Finally, we remark that, at each level of the search tree, we just replace an inequality constraint on the sum of the items with another one. Therefore, we solve model CKP_b with a single additional inequality constraint in each of the branches. Generally speaking, adding inequalities to a problem may increase the computational effort required to solve it. Given the characteristics of model CKP_b outlined in the previous sub–section 3.4.1, it is reasonable not to expect a meaningful increase of the computational cost when a single inequality constraint is added. Our computational experiments confirm this statement.

3.4.3 Solving the sub–problems

The last step consists in solving by means of an ILP solver several times model $CKP_a(S)$ for different values of S in constraint (3.15). Then, the best solution among the solutions of the various sub-problems yields the optimal solution of CKP.

The pseudo code of the approach is provided in Algorithm 4.

Algorithm 4 Exact approach

Input: CKP instance.
$LB' \leftarrow$ Apply the reduction procedure;
$(\tilde{UB}, \tilde{x}) \leftarrow$ solve model CKP_b;
$S \leftarrow \sum_{i=1}^{n} \tilde{x_i}$;
$LB \leftarrow$ solve model $CKP_a(S)$;
$Best \leftarrow \max\{LB, LB'\}$;
check:=TRUE, $s_{left} := S$;
while *check* **do** ▷ Left branch
    $s_{left} = s_{left} - 1$;
    $(\hat{UB}, \hat{x}) \leftarrow$ solve model $CKP_b \bigcap \sum_{i=1}^{n} x_i \leq s_{left})$;
    if $\hat{UB} < Best + 1$ **then**
        check:=FALSE;
    else
        add $\sum_{i=1}^{n} x_i = s_{left}$ to the sub–problems list;
    end if
end while
check:=TRUE, $s_{right} := S$;
while *check* **do** ▷ Right branch
    $s_{right} = s_{right} + 1$;
    $(\hat{UB}, \hat{x}) \leftarrow$ solve model $CKP_b \bigcap \sum_{i=1}^{n} x_i \geq s_{right}$;
    if $\hat{UB} < Best + 1$ **then**
        check:=FALSE;
    else
        add $\sum_{i=1}^{n} x_i = s_{right}$ to the sub–problems list;
    end if
end while
for all s in the sub–problems list **do** ▷ Solve sub–problems
    $LB^{sub} \leftarrow$ solve model $CKP_a(S)$ with $S = s$;
    if $LB^{sub} > Best$ **then**
        $Best \leftarrow LB^{sub}$;
    end if
end for
return $Best$;

3.5 Computational results

All tests have been conducted on a Intel i5 CPU @ 3.3 GHz with 4 GB of Ram. The ILP solver used was CPLEX 12.5 and the code has been implemented in C++ programming language. The parameters of CPLEX 12.5 were set to their default values.

3.5.1 Results for CKP

In preliminary testing, we generated instances according to the scheme provided in literature in [30], [71] and [88]. The instances are uncorrelated with the profits of items randomly ranging from 1 to a parameter p, while the weights of items vary from 1 to a parameter w. The capacity values of $B(\cdot)$ are constructed by randomly generating h numbers in a range from 1 to c, sorting these values in non–increasing order and assigning them to $B(1),\ldots,B(\mathrm{h})$ and $B(\mathrm{j}) = 0$ for $j = h+1,\ldots,n$. We generated instances with 1000 items (the largest size considered up to now in literature) by using the same parameters of [88], that is $p = 300$, $w = 1000$ and $c = 1000$. CPLEX 12.5 applied to model CKP_a solves to optimality all instances with 1000 items in less than a second in the worst case.

We considered then much larger size and challenging instances involving different correlations between profits and weights of the items . More precisely, we investigated 9 types of correlations as proposed in [57]. We considered instances with up to 100000 items with profits and weights ranging in $[1,1000]$. We chose the value of parameter c proportional to the sum of the weights, in particular we set c equal to $0.1\sum w$, and we considered three possible values of h proportional to the number of items ($0.1n$, $0.2n$, $0.5n$). The results obtained by CPLEX 12.5 on model CKP_a are reported in Table 3.1 and Table 3.2 where, for each value of n, are reported the average and maximum CPU time and the number of optima reached within a time limit of 600 seconds. The average CPU times consider also the cases where the optimum is not found.

We first notice that CPLEX 12.5 applied to model CKP_a is in general well performing on CKP solving to optimality most of the instances, but it does not

CPLEX 12.5	n	10000			20000			30000			40000			50000		
Correlation	h/n	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
Uncorr.	0.10	1.30	2.82	20	2.20	5.77	20	6.83	15.09	20	23.11	189.23	20	22.16	139.92	20
	0.20	1.39	4.06	20	5.35	30.76	20	7.05	16.22	20	16.22	121.40	20	27.58	132.76	20
	0.50	1.25	2.54	20	9.80	87.81	20	34.58	180.24	20	29.89	200.26	20	50.34	600.00	19
Weakly corr.	0.10	1.05	7.41	20	1.95	7.15	20	10.76	65.10	20	47.73	320.13	20	74.74	457.27	20
	0.20	3.08	23.17	20	6.85	44.71	20	20.10	129.65	20	24.27	128.84	20	155.46	600.00	19
	0.50	3.03	28.30	20	8.18	83.27	20	39.70	220.93	20	147.79	600.00	18	297.86	600.00	13
Strongly corr.	0.10	0.35	0.92	20	0.36	1.23	20	0.74	2.75	20	1.04	4.21	20	1.38	5.04	20
	0.20	0.28	0.87	20	0.72	2.61	20	0.82	2.04	20	2.70	5.83	20	4.02	25.24	20
	0.50	0.17	0.42	20	0.20	0.30	20	0.56	3.60	20	0.47	1.33	20	0.72	4.13	20
Inverse	0.10	0.26	0.94	20	0.32	1.22	20	0.73	3.14	20	0.95	3.42	20	1.00	4.09	20
strongly corr.	0.20	0.32	1.08	20	0.45	1.26	20	0.63	3.42	20	1.80	5.74	20	2.53	10.06	20
	0.50	0.41	1.45	20	1.32	5.98	20	2.08	5.88	20	3.93	8.80	20	3.84	9.50	20
Almost	0.10	0.38	1.08	20	0.80	2.39	20	1.56	8.10	20	1.33	4.34	20	1.88	7.69	20
strongly corr.	0.20	0.46	1.42	20	1.08	5.13	20	1.30	5.68	20	3.87	15.90	20	4.80	19.86	20
	0.50	0.42	2.57	20	0.63	3.71	20	0.65	1.36	20	1.14	5.82	20	0.85	2.32	20
Subset–sum	0.10	0.24	1.03	20	0.27	1.01	20	0.85	2.43	20	1.44	6.30	20	1.02	4.27	20
	0.20	0.28	1.39	20	0.77	2.23	20	1.56	7.91	20	2.92	8.32	20	3.05	7.33	20
	0.50	0.51	1.93	20	1.07	3.00	20	2.14	5.74	20	4.93	18.55	20	4.84	12.18	20
Even–odd	0.10	0.20	0.61	20	0.30	0.81	20	0.67	2.23	20	1.22	2.89	20	1.30	3.54	20
subset sum	0.20	0.28	1.17	20	0.80	2.84	20	1.65	5.98	20	2.00	6.15	20	4.25	20.83	20
	0.50	1.10	4.13	20	1.28	2.90	20	1.54	3.57	20	4.66	19.24	20	3.94	13.21	20
Even–odd	0.10	0.24	0.69	20	0.38	1.45	20	0.74	2.03	20	1.04	3.82	20	1.03	4.48	20
strongly corr.	0.20	0.32	1.12	20	0.92	2.14	20	1.60	5.54	20	2.54	6.38	20	2.50	7.64	20
	0.50	0.16	0.45	20	0.23	0.44	20	0.41	2.57	20	0.45	0.97	20	0.67	4.32	20
Uncorr. with	0.10	0.20	0.22	20	0.37	0.75	20	0.53	0.76	20	0.64	0.94	20	1.14	2.71	20
similar weights	0.20	0.21	0.33	20	0.38	0.56	20	0.67	1.08	20	0.95	1.70	20	1.34	2.28	20
	0.50	0.29	0.66	20	0.44	0.56	20	0.73	1.09	20	1.20	1.92	20	1.60	2.29	20

Table 3.1 CPLEX results on CKP instances with different correlations between profits and weights. Number of items from 10000 to 50000.

manage to solve weakly correlated and uncorrelated instances with more than 30000 variables. CPLEX 12.5 fails to reach the optimum within 600 seconds in several cases as the size of the instances increases. The uncorrelated and weakly correlated instances turned out to be the even more challenging than the strongly correlated ones as considered in the schemes in literature. The other types of correlation appear to be relatively easy to solve to optimality even for instances with 100000 items.

We focused then on the first two correlation types and apply the reduction procedure presented in Section 3.3 to instances from 30000 with up to 100000 items. After that we launched CPLEX 12.5 on the reduced problems. The results are reported in Table 3.3 where, for each value of n the average and maximum CPU time within a time limit of 600 seconds are reported. We also report the minimum percentages of the sums of items (namely the y_j variables) further discarded with respect to the feasible sums obtained after the application of Property 1.

The procedure shows up to be able of drastically reducing the solution space of the problem. It allows to discard at least the 87.9% of the sums of the items for all the instances considered. As one may expect, the method enhances the performance of the ILP solver, which is able to solve to optimality all the

CPLEX 12.5	n	60000			70000			80000			90000			100000		
Correlation	h/n	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
Uncorr.	0.10	58.71	384.68	20	63.58	600.00	19	126.11	600.00	19	138.27	600.00	18	271.50	600.00	13
	0.20	73.07	504.73	20	152.01	600.00	19	186.29	600.00	18	184.75	600.00	18	414.78	600.00	10
	0.50	95.68	600.00	19	138.24	600.00	17	128.22	600.00	18	250.26	600.00	13	339.71	600.00	12
Weakly corr.	0.10	154.73	600.00	17	246.25	600.00	16	299.88	600.00	15	415.56	600.00	11	408.88	600.00	9
	0.20	200.87	600.00	16	381.67	600.00	11	366.69	600.00	12	429.12	600.00	10	363.00	600.00	10
	0.50	204.85	600.00	16	311.59	600.00	13	432.05	600.00	8	417.84	600.00	9	427.79	600.00	8
Strongly corr.	0.10	2.16	9.84	20	3.19	11.56	20	3.89	12.32	20	5.66	18.53	20	4.36	15.23	20
	0.20	4.09	15.82	20	4.80	11.97	20	6.32	16.18	20	6.71	16.90	20	7.95	63.49	20
	0.50	0.71	2.28	20	0.89	5.12	20	0.63	1.17	20	0.79	1.42	20	0.77	0.81	20
Inverse	0.10	1.47	5.48	20	2.72	7.66	20	3.31	8.99	20	6.17	26.26	20	5.12	13.11	20
strongly corr.	0.20	3.78	17.21	20	5.17	13.26	20	4.60	19.05	20	5.78	14.45	20	6.39	20.76	20
	0.50	3.19	14.99	20	7.30	26.01	20	7.82	36.75	20	8.00	28.53	20	12.84	59.70	20
Almost	0.10	4.32	10.97	20	5.18	18.00	20	6.57	22.31	20	9.22	19.39	20	7.49	18.13	20
strongly corr.	0.20	5.37	16.79	20	5.37	20.95	20	8.14	26.19	20	6.80	29.13	20	17.65	71.34	20
	0.50	0.94	1.33	20	1.70	7.10	20	1.73	5.40	20	2.88	13.40	20	2.59	10.16	20
Subset–sum	0.10	2.65	9.59	20	2.53	6.66	20	2.54	7.69	20	6.63	21.95	20	4.22	11.54	20
	0.20	4.19	10.03	20	3.88	10.25	20	5.90	12.90	20	5.94	16.49	20	7.72	32.89	20
	0.50	5.16	14.90	20	7.99	25.13	20	7.62	31.20	20	7.19	22.65	20	18.97	95.43	20
Even–odd	0.10	2.40	5.41	20	2.67	7.11	20	3.61	9.78	20	4.61	22.31	20	5.45	15.93	20
subset sum	0.20	3.98	27.55	20	4.83	31.61	20	7.29	21.89	20	5.19	22.43	20	10.63	30.30	20
	0.50	7.40	24.93	20	6.25	15.34	20	7.25	18.25	20	8.52	28.55	20	11.49	44.24	20
Even–odd	0.10	1.43	6.43	20	3.07	9.53	20	3.40	18.88	20	4.73	24.32	20	6.61	18.27	20
strongly corr.	0.20	4.72	19.81	20	6.19	20.67	20	6.60	22.60	20	8.39	30.47	20	10.85	47.30	20
	0.50	0.58	1.16	20	0.56	0.90	20	0.72	1.55	20	0.79	2.37	20	1.78	11.01	20
Uncorr. with	0.10	1.43	4.38	20	1.61	3.92	20	2.02	4.91	20	3.00	12.36	20	2.57	5.76	20
similar weights	0.20	1.74	2.92	20	2.38	4.04	20	3.07	6.26	20	2.99	5.62	20	3.46	5.80	20
	0.50	1.98	2.90	20	2.82	4.68	20	2.94	4.65	20	3.37	5.90	20	3.65	5.57	20

Table 3.2 CPLEX results on CKP instances with different correlations between profits and weights. Number of items from 60000 to 100000.

instances with up to $n = 100000$ in less than 102 seconds approximately in the worst case (for a weakly correlated instance with 90000 items).

In Table 3.4 are reported the related results of the exact approach of Section 3.4: The results are similar to those of Table 3.3. We notice, however, that the exact approach generally performs slightly better in the worst-case instances and manages to solve to optimality all instances requiring at most 82 seconds. As discussed in Section 3.5.3, the exact approach makes a difference in the performance for M-CKP, in particular in 2-CKP.

3.5.2 Comparison with the algorithms in literature

We remark that even though a direct comparison with the algorithms present in literature is not provided, we claim that our approach is supposed to be significantly superior to these algorithms. Let us consider first the approach proposed in [71]. The experimental results were carried out approximately 20 years ago, thus it is very hard to perform a comparison between the computers. Nevertheless, that approach has a complexity $\mathcal{O}(nKb_1)$, where K indicates the number of possible capacity values in an optimal solution. In our computational experiments we have $b_1 \geq 0.1 \sum w_i$, $K \geq 0.05n$. As, $\sum w_i \cong 500n$, roughly

Reduction proc. + CPLEX 12.5	n	30000		40000		50000		60000	
Correlation	h/n	Average time (s)	Max time (s)	Average time (s)	Max time (s)	Average time (s)	Max time (s)	Average time (s)	Max time (s)
Uncorr.	0.10	3.02	5.99	3.44	5.71	5.99	9.88	6.57	13.95
	0.20	3.46	7.66	4.22	8.10	5.25	13.81	5.65	11.97
	0.50	3.26	7.94	4.32	8.80	5.78	13.34	8.05	23.70
Weakly corr.	0.10	2.37	3.64	3.82	12.26	4.58	6.30	6.95	20.76
	0.20	2.87	6.82	4.33	7.18	5.43	7.75	6.54	11.20
	0.50	5.15	17.30	6.91	17.86	11.45	42.79	9.75	27.85
Min % of sums of items discarded		88.0		88.9		87.9		91.0	

	n	70000		80000		90000		100000	
Correlation	h/n	Average time (s)	Max time (s)	Average time (s)	Max time (s)	Average time (s)	Max time (s)	Average time (s)	Max time (s)
Uncorr.	0.10	8.59	17.52	9.26	16.47	11.00	23.26	13.16	37.60
	0.20	9.22	15.88	11.67	24.52	10.70	23.76	14.33	32.64
	0.50	8.93	15.46	13.16	32.67	14.80	28.44	14.03	29.61
Weakly corr.	0.10	7.59	11.72	8.95	16.41	10.92	19.91	11.48	17.41
	0.20	8.65	13.64	10.93	19.33	11.37	20.53	13.59	28.13
	0.50	21.11	90.46	21.82	58.89	24.08	101.93	22.69	93.57
Min % of sums of items discarded		89.3		91.2		89.8		92.1	

Table 3.3 CKP uncorrelated and weakly correlated instances. Results obtained by applying the reduction procedure before running CPLEX 12.5.

speaking, we have $b_1 \cong 50n$ and $\mathcal{O}(nKb_1) = \mathcal{O}(2.5n^3)$. If we consider the largest instances handled by our approaches, i.e. $n = 100000$, we have $2.5n^3 = 2.5 * 10^{15}$. Even with a processor running at 10GHz, namely more than three times faster than our machine, this would induce more than 10^5 seconds of computation time, while we solve to optimality all instances in less than 82 seconds.

Let now consider the algorithm in [88]. The largest instances considered are with 1000 items and they are solved in more than 1000 seconds on a computer with a Celeron 300A CPU. We solved the same kind of instances in less than 1 second on a machine which is approximately 11 times faster[1]. We may reasonably assume that the differences in the performance of the algorithms would remain significant even if they were launched on the same machine.

3.5.3 Results for M-CKP

We consider M-CKP involving up to 5 capacity constraints. We first present the results for 2-CKP, which is a generalization of 2-KP, the well–known 2-constraint variant of the standard KP (see, e.g., [63]). We conducted a broad

[1]We consulted the following website for comparing the CPUs: http://cpuboss.com.

Exact approach	n	30000		40000		50000		60000	
Correlation	h/n	Average time (s)	Max time (s)	Average time (s)	Max time (s)	Average time (s)	Max time (s)	Average time (s)	Max time (s)
Uncorr.	0.10	3.75	4.99	5.06	6.77	7.56	10.25	8.94	12.76
	0.20	3.62	4.59	5.91	8.49	7.39	10.67	8.69	11.72
	0.50	3.87	5.48	5.53	7.11	7.81	10.59	9.92	13.92
Weakly corr.	0.10	3.42	4.26	5.73	12.22	7.00	8.11	9.09	10.98
	0.20	3.75	4.60	6.03	6.91	7.70	9.06	9.52	10.78
	0.50	4.89	6.96	8.36	20.12	10.92	16.68	13.26	40.61

	n	70000		80000		90000		100000	
Correlation	h/n	Average time (s)	Max time (s)	Average time (s)	Max time (s)	Average time (s)	Max time (s)	Average time (s)	Max time (s)
Uncorr.	0.10	11.95	15.30	13.24	18.31	16.60	22.71	18.50	24.26
	0.20	12.45	17.94	14.87	19.16	16.37	21.68	17.88	26.83
	0.50	11.55	16.36	14.93	22.71	19.12	25.33	20.53	24.93
Weakly corr.	0.10	10.63	12.56	12.57	15.40	15.17	23.26	16.32	24.37
	0.20	11.90	14.48	14.21	19.10	16.04	21.09	17.58	23.81
	0.50	16.59	28.86	22.72	50.50	23.16	48.28	26.89	82.09

Table 3.4 CKP uncorrelated and weakly correlated instances. Results obtained by applying the exact approach.

computational analysis by testing different sets of instances according to a scheme provided in the literature for the the two-dimensional KP. The results underline a remarkable effectiveness of our exact approach which turns out to outperform significantly both CPLEX 12.5 standalone and CPLEX 12.5 launched in cascade after the application of the reduction procedure. Then, we present the results for the other multidimensional variants. Our approach still globally outperforms the competitors, even though we notice a relevant decrease in the size of the instances solved to optimality within the time limit considered.

2-CKP

In 2-CKP, we considered the generation scheme for the instances of the two-dimensional KP provided in literature by [63], where different sets of instances are analyzed. In instances labeled as "B(R)" the weights are randomly generated from the uniform distribution [1, R], with R being an arbitrary parameter. The profit of each item is equal to one half of the sum of its weights plus r, with r randomly generated from the uniform distribution [1, $R/2$]. The capacity value of each constraint is equal to $0.25\sum w_i$ in B1, $0.5\sum w_i$ in B2 and $0.75\sum w_i$ in B3. In classes A1, A2, and A3 different correlation between profits and weights

of items are considered (see [63] for a detailed description). The capacity value of each constraint is equal to $0.5\sum w_i$.

We generated instances for 2-CKP with parameter R fixed to 1000 and by constructing two non-increasing functions of the capacity values. Similarly to CKP, the capacity function of each constraint is built by generating h numbers in a range from 1 to a parameter c. Considering how the instances of 2-KP problem have been generated, we set the parameter c equal to $0.4\sum w_i$ for class B1, $0.8\sum w_i$ for classes A1, A2, A3 , B2 and equal to $\sum w_i$ for class B3 respectively. Two values of h proportional to the number of items have been considered ($0.25n$, $0.5n$). We generated instances from 10000 to 100000 items and batches with 20 instances within each category. Table 3.5 summarizes the performance of CPLEX 12.5 standalone, CPLEX 12.5 launched in cascade after the application of the reduction procedure and our exact approach for the various classes of instances. For each class, the average and maximum CPU time within a time limit of 600 seconds and the number of optima reached over 400 instances are reported. The average times include also the cases where the optimum is not reached. Detailed results for different sets of instances are reported in Tables 3.6–3.11.

	CPLEX 12.5			Reduction proc. + CPLEX 12.5			Exact approach		
Class	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
A1	437.82	600.00	160	103.92	600.00	368	16.2	69.25	400
A2	255.03	600.00	279	13.19	88.39	400	9.82	34.51	400
A3	306.40	600.00	242	15.76	129.01	400	11.52	41.29	400
B1	474.64	600.00	123	167.93	600.00	334	27.86	121.96	400
B2	447.12	600.00	150	104.10	600.00	381	21.49	96.99	400
B3	459.76	600.00	147	97.51	600.00	380	19.50	80.04	400

Table 3.5 2-CKP benchmark instances time (s) and number of optima for each class, average over 400 instances.

Class	**A1**	CPLEX 12.5			Reduction proc. + CPLEX 12.5			Exact approach		
n	***h/n***	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
10000	0.25	22.40	57.38	20	2.79	8.55	20	1.97	3.00	20
	0.50	53.66	141.77	20	3.07	7.01	20	2.40	4.05	20
20000	0.25	125.41	336.24	20	25.64	215.61	20	4.01	5.96	20
	0.50	309.35	600.00	15	29.16	283.47	20	4.99	6.76	20
30000	0.25	282.70	600.00	19	13.54	82.29	20	6.81	9.39	20
	0.50	479.90	600.00	8	74.27	544.22	20	8.58	13.31	20
40000	0.25	416.92	600.00	11	40.35	422.31	20	8.18	10.56	20
	0.50	477.06	600.00	7	35.54	210.82	20	9.71	15.09	20
50000	0.25	493.69	600.00	6	105.85	600.00	17	11.97	16.65	20
	0.50	519.76	600.00	6	148.85	600.00	18	13.79	18.00	20
60000	0.25	533.41	600.00	3	95.20	600.00	18	14.35	27.58	20
	0.50	487.57	600.00	6	179.31	600.00	16	18.22	23.10	20
70000	0.25	582.27	600.00	3	95.36	600.00	19	17.06	23.62	20
	0.50	529.16	600.00	4	208.58	600.00	16	21.02	32.65	20
80000	0.25	571.13	600.00	4	117.22	600.00	19	21.08	28.91	20
	0.50	556.46	600.00	2	142.27	600.00	18	24.40	34.24	20
90000	0.25	586.86	600.00	1	209.55	600.00	15	28.45	38.77	20
	0.50	560.31	600.00	2	119.01	600.00	19	36.20	58.33	20
100000	0.25	594.51	600.00	1	170.75	600.00	17	30.10	43.54	20
	0.50	573.96	600.00	2	262.01	600.00	16	41.26	69.25	20

Table 3.6 2-CKP benchmark instances (class A1) time (s) and number of optima, average over 20 instances.

Class	**A2**	CPLEX 12.5			Reduction proc. + CPLEX 12.5			Exact approach		
n	***h/n***	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
10000	0.25	6.12	19.28	20	1.08	1.58	20	1.06	1.39	20
	0.50	7.09	28.14	20	1.13	1.93	20	1.21	1.55	20
20000	0.25	18.99	78.44	20	2.04	3.96	20	2.20	2.89	20
	0.50	38.97	361.75	20	2.47	4.59	20	2.81	3.79	20
30000	0.25	51.11	144.38	20	4.45	8.44	20	4.02	5.34	20
	0.50	137.45	600.00	19	4.01	8.69	20	4.04	5.90	20
40000	0.25	166.60	600.00	17	6.88	10.47	20	5.37	8.74	20
	0.50	215.51	600.00	17	7.73	12.09	20	5.91	8.19	20
50000	0.25	154.89	600.00	17	8.48	13.28	20	7.05	9.10	20
	0.50	205.25	600.00	16	9.99	16.51	20	7.68	10.53	20
60000	0.25	298.21	600.00	12	11.88	19.80	20	9.70	16.19	20
	0.50	247.63	600.00	16	10.45	18.71	20	9.48	12.31	20
70000	0.25	324.79	600.00	11	17.53	88.39	20	11.89	15.68	20
	0.50	304.08	600.00	15	17.67	46.08	20	13.71	20.61	20
80000	0.25	427.37	600.00	10	17.78	37.96	20	14.89	20.45	20
	0.50	452.50	600.00	8	24.55	47.77	20	18.50	30.87	20
90000	0.25	448.37	600.00	9	20.70	41.59	20	15.85	25.01	20
	0.50	520.78	600.00	5	35.28	59.56	20	20.90	32.54	20
100000	0.25	492.48	600.00	6	26.36	52.03	20	18.79	34.01	20
	0.50	582.40	600.00	1	33.26	82.66	20	21.43	34.51	20

Table 3.7 2-CKP benchmark instances (class A2) time (s) and number of optima, average over 20 instances.

Class	**A3**	CPLEX 12.5			Reduction proc. + CPLEX 12.5			Exact approach		
n	***h/n***	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
10000	0.25	5.79	31.95	20	0.85	1.90	20	1.11	1.55	20
	0.50	13.47	53.74	20	1.57	4.95	20	1.28	1.64	20
20000	0.25	20.45	73.04	20	2.79	14.37	20	2.43	3.45	20
	0.50	76.24	375.77	20	3.38	7.01	20	2.94	4.38	20
30000	0.25	72.88	374.60	20	3.38	7.39	20	3.71	5.13	20
	0.50	137.45	518.16	20	7.24	35.66	20	4.72	6.27	20
40000	0.25	118.27	383.17	20	5.20	12.43	20	5.30	7.21	20
	0.50	193.50	600.00	16	8.99	29.50	20	6.75	9.05	20
50000	0.25	267.41	600.00	15	9.11	30.22	20	7.21	9.94	20
	0.50	405.46	600.00	10	14.31	56.82	20	9.52	12.45	20
60000	0.25	301.67	600.00	14	13.22	32.51	20	10.44	14.12	20
	0.50	403.70	600.00	11	11.25	24.31	20	11.47	14.15	20
70000	0.25	384.41	600.00	10	15.22	40.42	20	12.06	20.20	20
	0.50	568.97	600.00	2	27.68	112.80	20	16.06	23.60	20
80000	0.25	456.37	600.00	8	18.88	61.09	20	14.97	22.09	20
	0.50	560.58	600.00	2	37.32	93.32	20	21.85	30.26	20
90000	0.25	513.64	600.00	5	28.87	84.02	20	18.83	25.44	20
	0.50	540.57	600.00	3	41.81	129.01	20	25.65	35.66	20
100000	0.25	521.56	600.00	4	31.07	126.72	20	22.72	27.85	20
	0.50	565.66	600.00	2	32.98	76.85	20	31.46	41.29	20

Table 3.8 2-CKP benchmark instances (class A3) time (s) and number of optima, average over 20 instances.

Class	**B1**	CPLEX 12.5			Reduction proc. + CPLEX 12.5			Exact approach		
n	***h/n***	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
10000	0.25	96.08	289.27	20	9.46	61.76	20	4.64	20.28	20
	0.50	169.03	494.10	20	81.99	495.04	20	7.07	10.42	20
20000	0.25	280.67	600.00	18	25.17	203.95	20	5.38	7.64	20
	0.50	387.24	600.00	11	90.26	600.00	19	9.10	16.55	20
30000	0.25	433.28	600.00	11	77.82	600.00	19	8.28	12.09	20
	0.50	533.96	600.00	5	179.77	600.00	16	14.39	25.58	20
40000	0.25	455.70	600.00	6	121.20	600.00	18	13.18	18.78	20
	0.50	434.39	600.00	7	162.36	600.00	16	24.01	38.77	20
50000	0.25	498.39	600.00	4	190.58	600.00	16	19.33	31.82	20
	0.50	543.23	600.00	3	196.05	600.00	15	22.61	34.16	20
60000	0.25	578.51	600.00	1	316.58	600.00	13	22.40	34.29	20
	0.50	553.92	600.00	2	157.07	600.00	17	32.96	68.50	20
70000	0.25	524.73	600.00	5	193.30	600.00	16	27.14	41.73	20
	0.50	508.35	600.00	5	183.25	600.00	16	42.15	73.43	20
80000	0.25	547.68	600.00	3	123.87	600.00	18	33.68	47.96	20
	0.50	573.03	600.00	1	254.80	600.00	14	51.13	84.33	20
90000	0.25	574.59	600.00	1	198.53	600.00	15	40.45	61.28	20
	0.50	600.00	600.00	0	273.88	600.00	14	65.00	121.96	20
100000	0.25	600.00	600.00	0	293.89	600.00	15	47.45	61.31	20
	0.50	600.00	600.00	0	228.82	600.00	17	66.94	94.72	20

Table 3.9 2-CKP benchmark instances (class B1) time (s) and number of optima, average over 20 instances.

Class	**B2**	CPLEX 12.5			Reduction proc. + CPLEX 12.5			Exact approach		
n	***h/n***	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
10000	0.25	31.68	101.92	20	10.14	140.31	20	2.20	4.48	20
	0.50	116.79	280.82	20	28.52	157.05	20	4.21	9.41	20
20000	0.25	159.51	383.57	20	17.07	122.68	20	4.24	6.37	20
	0.50	348.15	600.00	14	23.78	152.01	20	6.80	9.77	20
30000	0.25	341.16	600.00	14	21.64	167.44	20	6.57	10.14	20
	0.50	512.97	600.00	6	53.41	399.86	20	10.06	14.95	20
40000	0.25	417.21	600.00	11	34.57	294.82	20	10.67	19.09	20
	0.50	484.35	600.00	6	170.43	600.00	18	14.63	22.18	20
50000	0.25	470.93	600.00	9	46.88	489.99	20	14.35	21.84	20
	0.50	486.17	600.00	5	176.01	600.00	16	21.58	29.17	20
60000	0.25	522.39	600.00	3	86.90	490.26	20	20.33	30.23	20
	0.50	577.96	600.00	1	238.53	600.00	17	27.02	40.76	20
70000	0.25	547.93	600.00	3	95.64	600.00	19	23.45	40.17	20
	0.50	580.36	600.00	1	185.45	600.00	18	33.15	46.93	20
80000	0.25	534.30	600.00	5	94.09	377.10	20	26.01	32.48	20
	0.50	552.29	600.00	2	173.76	600.00	18	37.40	52.42	20
90000	0.25	534.29	600.00	6	151.51	600.00	19	31.79	51.75	20
	0.50	568.92	600.00	2	138.76	476.66	20	44.43	64.90	20
100000	0.25	600.00	600.00	0	179.42	600.00	18	38.40	50.50	20
	0.50	555.01	600.00	2	155.44	600.00	18	52.46	96.99	20

Table 3.10 2-CKP benchmark instances (class B2) time (s) and number of optima, average over 20 instances.

Class	**B3**	CPLEX 12.5			Reduction proc. + CPLEX 12.5			Exact approach		
n	***h/n***	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
10000	0.25	28.90	100.20	20	7.32	71.07	20	2.36	4.46	20
	0.50	78.07	228.54	20	6.72	33.74	20	3.49	5.43	20
20000	0.25	189.62	484.35	20	13.33	66.27	20	4.07	5.68	20
	0.50	364.75	600.00	14	71.34	585.30	20	5.76	7.93	20
30000	0.25	275.89	600.00	17	20.64	155.33	20	6.44	10.41	20
	0.50	513.19	600.00	4	58.14	419.84	20	8.68	12.36	20
40000	0.25	426.72	600.00	12	59.67	339.53	20	9.86	17.30	20
	0.50	485.34	600.00	6	151.04	600.00	16	12.69	19.33	20
50000	0.25	537.35	600.00	7	61.75	600.00	19	13.00	16.43	20
	0.50	570.78	600.00	3	55.98	458.02	20	15.74	21.93	20
60000	0.25	541.41	600.00	5	79.37	346.02	20	18.23	28.66	20
	0.50	589.43	600.00	3	204.84	600.00	15	23.89	41.11	20
70000	0.25	561.97	600.00	4	114.24	560.76	20	21.48	34.76	20
	0.50	562.00	600.00	4	105.29	600.00	18	28.79	46.04	20
80000	0.25	560.31	600.00	3	121.57	600.00	19	26.81	38.97	20
	0.50	588.91	600.00	1	148.16	600.00	18	34.62	44.40	20
90000	0.25	577.55	600.00	1	36.67	105.99	20	28.08	38.72	20
	0.50	548.52	600.00	2	148.78	600.00	19	38.94	57.77	20
100000	0.25	600.00	600.00	0	180.79	568.14	20	35.67	57.75	20
	0.50	594.49	600.00	1	304.66	600.00	16	51.48	80.04	20

Table 3.11 2-CKP benchmark instances (class B3) time (s) and number of optima, average over 20 instances.

From Table 3.5, we see that CPLEX 12.5 applied to model 2-CKP_a fails to reach all the optima running out of time in many cases. As shown in Tables 3.6–3.11, CPLEX 12.5 does not solve to optimality instances with more than 10000 variables. The application of the reduction procedure and then CPLEX 12.5 in cascade improves the performance but does not manage to solve to optimality all instances within the time limit. The proposed exact approach of Section 3.4 significantly outperforms the competitors being able to solve to optimality all instances with limited computational effort. The time required is 122 seconds approximately in the worst case (for an instance of class B1 with 90000 items). Our exact approach performs in general very well and confirms its effective contribution in solving a more challenging variant of the problem. Notice also that, as far as the exact approach is concerned, for 2-CKP the third main step was repeated in each instance at most four times (that is the number of times the procedure solved model $CKP_a(S)$). This is a remarkable strength

of the approach and probably crucial for solving large instances of the problem in limited time.

3-4-5-CKP

For these other M-CKP versions, we considered the generation scheme of M-KP provided in the literature by [19]. The weights of the items in each of the m capacity constraints are integer numbers randomly generated from the uniform distribution [1, 1000], while the profits of the items are correlated with their weights, namely $p_i = \sum_{m=1}^{M} w_i^{(m)}/M + q_i$ $(i = 1, ..., n)$, with q_i randomly generated from the uniform distribution [1, 500]. Similarly to the instances of 2-CKP, we generated M (with $M = 3, 4, 5$) non-increasing functions of the capacity values. The capacity function of each constraint is generated by extracting h numbers in a range from 1 to a parameter c. Considering the tightness ratios adopted in the instances of the multidimensional knapsack problem, we set the parameter c equal to $0.4\sum w_i$, $0.8\sum w_i$ and $\sum w_i$. We considered two values of h proportional to the number of items ($0.25n$, $0.5n$) and generated batches with 10 instances within each category. We tested instances with up 16000, 2000 and 1500 items in 3-CKP, 4-CKP and 5-CKP respectively. The profits and weights in the instances of 5-CKP with 500 items are those from the corresponding M-KP instances in [19]. The performance of CPLEX 12.5 standalone, CPLEX 12.5 launched in cascade after the application of the reduction procedure and our exact approach are reported in Table 3.12. Each entry in the table gives the average and maximum CPU time and the number of optima reached over 60 instances within a time limit of 3600 seconds. The average times include the cases where the optimum is not found.

From Table 3.12, we notice that the size of the instances solved to optimality by our exact approach is much more limited with respect to the results reached for CKP and 2-CKP. Our approach still outperforms the competitors on 3-CKP and 4-CKP and is comparable to CPLEX 12.5 launched in cascade after the application of the reduction procedure on 5-CKP (while CPLEX 12.5 reaches worse performance). Nevertheless, our approach is less effective as soon as the number of the items and the number of the capacity constraints increase. Our approach manages to solve to optimality all instances involving 12000, 1500

		CPLEX 12.5			Reduction proc. + CPLEX 12.5			Exact approach		
M	***n***	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
3	1000	5.85	42.51	60	3.59	22.14	60	3.02	17.63	60
	2000	27.65	203.24	60	11.89	111.54	60	7.47	55.27	60
	4000	385.78	3600.00	59	128.13	856.68	60	45.99	378.29	60
	8000	1259.09	3600.00	49	391.79	3600.00	57	153.14	1262.95	60
	12000	2158.14	3600.00	37	706.54	3600.00	54	285.94	1892.24	60
	16000	2573.36	3600.00	24	1024.03	3600.00	52	462.42	3600.00	59
4	500	3.73	21.20	60	2.85	13.57	60	2.90	16.46	60
	1000	40.49	339.27	60	26.38	210.32	60	19.82	142.15	60
	1500	196.77	1547.59	60	126.45	1512.27	60	76.95	797.80	60
	2000	503.25	3600.00	57	294.16	3600.00	58	224.49	3600.00	59
5	500	12.88	169.21	60	9.27	111.71	60	8.70	99.58	60
	1000	365.11	3600.00	59	234.36	2466.63	60	224.18	3528.01	60
	1500	1151.51	3600.00	48	820.82	3600.00	52	758.84	3600.00	52

Table 3.12 3-CKP, 4-CKP, 5-CKP benchmark instances time (s) and number of optima, average over 60 instances.

and 1000 items in 3-CKP, 4-CKP and 5-CKP respectively. As a matter of fact, these results were quite predictable, as we dealt with a generalization of M-KP, which is well–known to be hard to solve when the capacity constraints are more than two (see, e.g., [33] for a survey on the problem). Indeed, at the current state of the art, several instances of M-KP with 500 items and 30 constraints are still unsolved and the best performing recent approaches [9, 23, 56] do not reach in reasonable time the optimal solution value in several instances of M-KP with 500 items and 10 constraints only (just reference [9] reached all optima for the instances with 500 items and 10 constraints but required in the worst case several days of CPU). Our exact approach relies on solving in the first step a multidimensional knapsack problem with the additional constraint that the number of items placed into the knapsack is fixed. Thus, finding a solution in the first step turns out to be very demanding when M increases, while this task is accomplished quite easily by the modern ILP solvers in CKP and 2-CKP. This constitutes a bottleneck for solving the multidimensional variants of CKP and marks the limit of applicability of our exact method within reasonable computational times.

Statistical performance analysis

In order to highlight the performance of the reduction procedure and of the different steps of the exact approach, we report a statistical analysis of the results

obtained in the computational experiments. For each instance, we first consider the LP relaxation and the upper bound UB provided by its optimal solution. The quality of the upper bound is evaluated by computing the percentage gap between UB and the first feasible solution LB' ($LP\ Gap = 1 - \frac{LB'}{UB}$). Then, we consider the percentage of variables x_i fixed by applying (3.31) as well as the percentage of variables y_j which would be fixed to 0 through the reduced costs but without applying the reduction procedure. These outcomes provide further insights about the effectiveness of the reduction procedure. For this procedure, we report the percentage of variables y_j set to 0 and the percentage difference between the optimal solution z^* and the first solution LB' ($Opt\ Gap = 1 - \frac{LB'}{z^*}$). The computational times (% of the total CPU time) of the reduction procedure and each of the 3 steps of the exact approach are also reported. For the second step, we provide the percentage of the sub–problems left at the end of the step. Eventually, we indicate the average and maximum number of sub–problems solved in step 3. Table 3.13 summarizes the results in terms of average and standard deviation values for each of the M-CKP versions considered. The analysis considers the instances solved to optimality within the CPU time limit.

								Exact approach					
		LP Relaxation			Reduction Procedure			Step 1	Step 2		Step 3		
M-CKP		y_j Fixed (%)	x_i Fixed (%)	LP Gap (%)	Time (%)	y_j Fixed (%)	Opt Gap (%)	Time (%)	Time (%)	Sub-pb left (%)	Time (%)	Sub-pb solved	(Max) Sub-pb solved
2-CKP	Average	50.4	0.1	0.3	34.5	99.1	0.1	42.7	22.8	0.0	0.0	0.0	4
	Std. Dev.	30.5	1.8	0.3	12.2	0.6	0.1	13.7	6.5	0.0	0.5	0.1	
3-CKP	Average	45.8	0.3	0.8	3.6	97.6	0.2	92.6	3.6	0.0	0.3	0.1	16
	Std. Dev.	30.1	3.1	0.8	5.5	1.7	0.3	10.9	5.3	0.1	2.0	0.9	
4-CKP	Average	37.6	0.8	1.5	5.8	96.1	0.5	88.3	5.0	0.1	1.0	0.3	6
	Std. Dev.	29.1	5.3	1.2	8.4	2.2	0.5	15.7	7.2	0.3	4.8	0.8	
5-CKP	Average	30.3	0.0	1.7	5.6	96.1	0.6	87.6	4.2	0.2	2.7	0.3	4
	Std. Dev.	26.2	0.0	1.2	8.2	1.9	0.6	16.9	6.4	0.4	9.6	0.7	

Table 3.13 Statistical performance analysis of the reduction procedure and of the exact approach for M-CKP.

The results in Table 3.13 illustrate the impressive effectiveness and the robustness of the reduction procedure in limiting the solution space of the problems. The procedure discards at least 96.1% of the y_j variables on average. Besides, the fluctuations around the average values are limited. Despite the overall good quality of the first solutions and of the LP bounds, on average the y_j variables which would be fixed to 0 by exploiting only the LP relaxation are much smaller. Large standard deviation values denote a quite high variability of the results as well. In general, the number of x_j variables fixed to their

optimal values is low. Moreover, as previously outlined for 2-CKP, the statistics indicate that the first two steps of the exact approach are usually sufficient to solve to optimality the instances of the multidimensional variants of CKP. The sub–problems solved in step 3 are very few with a maximum value of 16 reached in an instance of 3-CKP with 4000 items. Finally, the results in column "Step 1" underline the drastic increase of the computational effort required in the first step of the approach when the capacity constraints are more than two.

Let us finally remark that in this chapter we presented an extensive computational analysis which broadened the previous analysis for CKP. We considered instances involving all classical correlations between profits and weights of the items and proposed a set of new benchmark instances for multidimensional variants with up to five capacity constraints. To get a broader picture of the effectiveness of our approaches, we may investigate in future research different generation schemes of the capacity values through non linear functions of the sum of the items. This would provide further insights about how the practical hardness of the problems could be related to the distribution of the capacity entries.

The 0–1 Penalized Knapsack Problem

4.1 Introduction

We consider the 0–1 Penalized Knapsack Problem (PKP), as introduced in [15]. PKP is a generalization of KP where each item has a profit, a weight and a penalty. The problem calls for maximizing the sum of the profits minus the greatest penalty value of the items included in a solution.

PKP may have applications in allocation resource problems with a bi–objective function involving the maximization of the sum of the profits against the minimization of the maximum value of a feature of interest. As an example, applications may derive in batch production systems where the processing time/cost of batches of products depends on the maximum processing time of each product. PKP may also occur as sub–problem within algorithmic frameworks designed for more complex problems. For instance, PKP arises as a pricing sub–problem in branch–and–price algorithms for the two–dimensional level strip packing problem in [54].

PKP is $\mathcal{NP}$–hard in the weak sense since it contains the standard KP as special case, namely when the penalties of the items are equal to zero, and it can solved by a pseudo–polynomial algorithm. In [15], a dynamic programming approach is recalled with time complexity $O(n^2c)$, with n and c being the number of items and the capacity of the knapsack respectively. Moreover, an exact algorithm is presented and successfully tested on instances with 1000 variables while running into difficulties on instances with 10000 variables. The approach relies on solving standard knapsack problems induced by choosing the item yielding the penalty value, denoted as *leading item*, and discarding

the items with higher penalties. The order of the sub–problems to be explored is greedily determined according to the upper bounds given by their linear relaxation.

The contribution of this work is twofold. At first, we provide insights into the problem and a characterization of its linear relaxation. We also present a surprising negative approximation result.
Secondly, we propose an exact approach that relies on a procedure narrowing the relevant range of penalties and on dynamic programming algorithms. A straightforward pseudopolynomial algorithm running with complexity $O(\max\{n \log n, nc\})$ is lined out. Then, as our major contribution, we devise a dynamic programming algorithm based on a core problem and the algorithmic framework proposed in [76].
We investigate the effectiveness of our approach on a large set of instances generated according to the literature and involving different types of correlations between profits, weights and penalties. The proposed approach turns out to be very effective in solving hard instances and compares favorably to both solver CPLEX 12.5 and the exact algorithm in [15], successfully solving all instances with up to 10000 items.

A journal version of the contribution of this chapter has been recently submitted. Part of the contents has been presented at AIRO 2015 conference. The chapter is organized as follows. In Section 4.2, the linear programming formulation of the problem is introduced. In Section 4.3, insights on the structure and properties of PKP are provided. We outline our new exact solution approach in Section 4.4 and discuss the computational results in Section 4.5.

4.2 Notation and problem formulation

In PKP a set of n items is given together with a knapsack with capacity c. Each item j has a non–negative integer weight w_j, a non–negative profit p_j and a non–negative penalty π_j. The problem calls for maximizing the total profit minus the greatest penalty value of the selected items without exceeding the knapsack capacity c.

In order to derive an ILP-formulation, we associate with each item j a binary variable x_j such that $x_j = 1$ iff item j is placed into the knapsack. Also, we associate a real variable Π with the decrease in the objective produced by the highest penalty value of the items placed in the knapsack. The problem can be formulated as follows:

(PKP):

$$\text{maximize} \quad \sum_{j=1}^{n} p_j x_j - \Pi \tag{4.1}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_j \leq c \tag{4.2}$$

$$\pi_j x_j \leq \Pi \qquad j = 1, \ldots, n \tag{4.3}$$

$$x_j \in \{0,1\} \qquad j = 1, \ldots, n \tag{4.4}$$

$$\Pi \in \mathbb{R} \tag{4.5}$$

(4.2) is the standard capacity constraint. Constraints (4.3) ensure that Π will carry the highest penalty value in any feasible solution of PKP; variable Π can be defined as real (constraint (4.5)) and will take in an optimal solution of model (PKP) one of the values π_j. The objective function (4.1) maximizes the sum of the profits minus the greatest penalty value of the selected items. We will denote the optimal objective function value by z^* and the item yielding the optimal penalty by j^*.

Henceforth, we will assume that items are sorted in decreasing order of penalties, i.e.

$$\pi_1 \geq \pi_2 \geq \ldots \geq \pi_n \tag{4.6}$$

Also, for any considered sub–problem PP the optimal objective function value will be written as $z(PP)$.

For further analysis, we will consider the penalty value Π as a fixed parameter and define $PKP(\Pi)$ as an instance of PKP where the penalty value in the objective function is fixed to Π. In the binary case, this simply means that all items j with $\pi_j > \Pi$ are eliminated from consideration and the remaining problem reduces to a standard 0–1 knapsack problem (KP). Notice that $PKP(\Pi)$ only needs to be considered at most for the n relevant choices $\Pi \in \{\pi_1, \ldots, \pi_n\}$. This implies that (PKP) can be solved to optimality by solving at most n

different 0-1 knapsack problems of type $PKP(\Pi)$ and taking the maximum objective function value. Clearly, this approach is not expected to be effective as soon as the number of items and thus of the penalty values increases.

4.3 Generalities and algorithmic insights

We provide here further insights on the structure and properties of PKP. We first show how the linear relaxation of the problem has a special structure and can be effectively solved. Then, we discuss an improved variant of the procedure proposed in [15] for computing upper bounds on the sub–problems induced by the selection of the leading item. We also derive a negative approximation result. Finally, we outline a basic dynamic programming algorithm with running time $O(\max\{n \log n, nc\})$.

4.3.1 Linear relaxation of PKP

In the linear relaxation of PKP, denoted PKP^{LP}, constraints (4.4) are replaced by the inclusion in the interval $[0, 1]$, i.e. items can be split and only a fractional part is packed. In this case a proportional part of the penalty applies. The optimal objective function value will be denoted by z^{LP}.

The LP-relaxation parametrically depending on Π will be denoted as $PKP^{LP}(\Pi)$. In the LP-relaxation for a given value Π, each variable x_j is upper bounded by the following expression:

$$x_j \leq \min\{1, \Pi/\pi_j\}. \tag{4.7}$$

This means that all items with a penalty exceeding the current Π, are reduced by scaling such that their penalty contribution is equal to Π. After imposing this bound the problem reduces again to an instance of KP (for fixed Π) for which the LP-relaxation is trivial.

As usually, the split item is the first item that would exceed the capacity if set to its upper bound according to (4.7). Considering the optimal solution value of $PKP^{LP}(\Pi)$ as a function in Π we get the following characterization.

Theorem 7. *$z^{LP}(\Pi)$ is a piecewise-linear concave function in Π consisting of at most $2n$ linear segments.*

Proof. Consider an arbitrary value of Π and the corresponding solution $x^{LP}(\Pi)$. Let S denote the set of items j with $x_j^{LP}(\Pi) = \Pi/\pi_j$, i.e. all items whose values are currently bounded by the considered penalty value Π. The current split item will be denoted as s.

We analyze the slope on the left-hand side of $(\Pi, z^{LP}(\Pi))$ by considering the change of the function implied by a decrease of the penalty bound from Π to $\Pi - \varepsilon$ for some small $\varepsilon > 0$. Formally, this change $\delta(\Pi)$ is given as follows:

$$\begin{aligned}\delta(\Pi) = z^{LP}(\Pi) - z^{LP}(\Pi-\varepsilon) &= -\varepsilon + \underbrace{\sum_{j\in S}\frac{\varepsilon}{\pi_j}p_j}_{\text{reduction of items in } S} - \underbrace{\frac{p_s}{w_s}\sum_{j\in S}\frac{\varepsilon}{\pi_j}w_j}_{\text{increase of split item}} \\ &= \varepsilon\left(-1+\sum_{j\in S}\frac{w_j}{\pi_j}\left(\underbrace{\frac{p_j}{w_j}-\frac{p_s}{w_s}}_{\geq 0}\right)\right) \qquad (4.8)\end{aligned}$$

This expression can be positive or negative, but it shows that $\delta(\Pi)$ is proportional to ε. Thus, in a neighborhood of $(\Pi, z^{LP}(\Pi))$ the function consists of a linear piece which will end in one of the following three cases:

1. $\Pi - \varepsilon = \pi_k$ for some $k \notin S$. This means that lowering Π, a new item is found for inclusion in S. Plugging in the extended set S in (4.8) will clearly increase the change $\delta(\Pi)$.

2. x_s reaches 1. This means that the split item becomes integral and item $s+1$ becomes the new split item. Thus, we replace $\frac{p_s}{w_s}$ by $\frac{p_{s+1}}{w_{s+1}}$ in (4.8) again implying an increase of $\delta(\Pi)$.

3. x_s reaches $(\Pi - \varepsilon)/\pi_s$. This means that the increase of the split item reaches the lowered penalty bound. In this case, s is included in S and $s+1$ becomes the new split item. Again, $\delta(\Pi)$ is increased by combining both of the above arguments.

Summarizing, we have shown that starting with an arbitrary value of Π and decreasing Π, $z^{LP}(\Pi)$ consists of a linear piece which ends with some $\Pi' \leq \Pi$ in

one of three possible configurations. The slope of this linear piece is given by $\frac{\delta(\Pi)}{\varepsilon}$. The preceding linear segment of $z^{LP}(\Pi')$ will have an *increased* change $\delta(\Pi')$ if Π' is further decreased. This means that the previous linear segment at $z^{LP}(\Pi')$ has a *larger slope* than $z^{LP}(\Pi)$. Thus, the slope of $z^{LP}(\Pi)$ is *decreasing* with increasing Π which proves the concavity of $z^{LP}(\Pi)$.

Starting the above procedure with $\Pi = \max_{j=1}^{n} \pi_j$ and reducing Π iteratively until $\Pi = 0$, it is clear that each item may cause the end of a linear segment of $z^{LP}(\Pi)$ at most twice: Once, by becoming a new split item and once by being included in set S. Each such event can occur at most once for each item. Therefore, there can be at most $2n$ linear pieces.

□

Proposition 4. *PKP^{LP} can be computed in $O(n \log n)$ time.*

Proof. Algorithmically, one can easily exploit the structure established in Theorem 7 by performing a binary search over Π to determine a maximum[1] of the concave function $z^{LP}(\Pi)$. For each query value Π, one can compute the split item in linear time and also assemble the corresponding set S in one pass through the set of items. Thus, for each query value Π the sign of the slope can be calculated from (4.8) in linear time.
Applying the binary search over all possible penalty values would yield a total running time of $O(n \log \pi_{\max})$ which is polynomial in the size of the (binary) encoded input, i.e. weakly polynomial. To obtain a strongly polynomial time algorithm whose running time depends only on the number of input values, we can first perform a binary search over all n values π_j and thus compute in $O(n \log n)$ time the interval of two consecutive penalties $[\pi_k, \pi_{k-1}]$ for some k (with $\pi_{k-1} \geq \pi_k$ according to (4.6)) which will include the optimal penalty value Π^{LP}.
Let us denote the optimal split item by s^{LP} and the split items associated with penalties π_{k-1} and π_k by s_{k-1} and s_k respectively. If we consider the items sorted by decreasing $\frac{p_j}{w_j}$, item s_{k-1} will precede item s_k in the ordering.
If $\pi_{k-1} = \pi_k$, clearly we have $s^{LP} = s_{k-1} = s_k$ and $\Pi^{LP} = \pi_{k-1} = \pi_k$. Otherwise,

[1] Note that the maximum is not necessarily unique, since there may exist a linear segment of $z^{LP}(\Pi)$ with slope 0.

we have to find s^{LP} in the interval $[s_{k-1}, s_k]$ and related Π^{LP} in the interval $[\pi_k, \pi_{k-1}]$.

Let us consider a generic item $\overline{s}$ as candidate for the split item. The best penalty Π associated with it can be calculated as follows. Given the interval $[\pi_k, \pi_{k-1}]$, we set $x_j = 1$ $(j = 1, \ldots, \overline{s}-1)$ if $\pi_j \leq \pi_k$, $x_j = \frac{\Pi}{\pi_j}$ otherwise. This implies that the weight and profit sums of the items preceding $\overline{s}$ are linear functions of Π in the form

$$\sum_{j=1}^{\overline{s}-1} w_j x_j = \gamma_1 \Pi + \gamma_2, \tag{4.9}$$

$$\sum_{j=1}^{\overline{s}-1} p_j x_j = \theta_1 \Pi + \theta_2, \tag{4.10}$$

with non-negative coefficients γ_1, γ_2, θ_1, θ_2 determined according to the above x_j setting. Item $\overline{s}$ can be the split item if and only if $\sum_{j=1}^{\overline{s}-1} w_j x_j < c$ and its value $x_{\overline{s}}$ fulfills the capacity (i.e. $x_{\overline{s}} = \frac{c - \sum_{j=1}^{\overline{s}-1} w_j x_j}{w_{\overline{s}}}$) while satisfying the bound (4.7). Correspondingly, the feasible interval of Π, denoted as $I_{\overline{s}}(\Pi)$, which allows $\overline{s}$ to be the split item is defined by the following system of inequalities:

$$I_{\overline{s}}(\Pi) := \begin{cases} \pi_k \leq \Pi \leq \pi_{k-1} \\ \gamma_1 \Pi + \gamma_2 \leq c - \varepsilon \\ \frac{c - \gamma_1 \Pi - \gamma_2}{w_{\overline{s}}} \leq \beta \text{ with } \beta = \begin{cases} 1 & \text{if } \pi_{\overline{s}} \leq \pi_k; \\ \frac{\Pi}{\pi_{\overline{s}}} & \text{otherwise} \end{cases} \end{cases} \tag{4.11}$$

Item $\overline{s}$ is a relevant candidate for the split item only if the corresponding interval $I_{\overline{s}}(\Pi)$ is non-empty. In such a case, the overall profit given by $\overline{s}$ as split item is

$$P_{\overline{s}}(\Pi) = \sum_{j=1}^{\overline{s}-1} p_j x_j + p_{\overline{s}} x_{\overline{s}} - \Pi = (\theta_1 - \frac{p_{\overline{s}}}{w_{\overline{s}}} \gamma_1 - 1)\Pi + \theta_2 + \frac{p_{\overline{s}}}{w_{\overline{s}}}(c - \gamma_2) \tag{4.12}$$

and will be maximized by choosing either the left extreme of $I_{\overline{s}}(\Pi)$ if the term $(\theta_1 - \frac{p_{\overline{s}}}{w_{\overline{s}}} \gamma_1 - 1) < 0$ or the right extreme otherwise.

Summarizing, the best penalty value associated with a candidate item can be computed in constant time if coefficients γ_1, γ_2, θ_1, θ_2 are given. Hence, we may

first consider item s_{k-1} as candidate for s^{LP} and compute related coefficients in (4.9)–(4.10) and penalty value. Then, we iteratively move to the next item after updating coefficients γ_1, γ_2, θ_1, θ_2 in one pass due the inclusion of the previous candidate item among items $j = 1,\ldots,\overline{s}-1$. After the evaluation of item s_k, the optimal split item s^{LP} and penalty Π^{LP} are returned. Since the execution time of this part is bounded by $O(n)$, the overall complexity for solving the LP relaxation is $O(n \log n)$. □

4.3.2 Computing upper bounds

A natural upper bound on PKP is given by z^{LP}. We may compute more involved upper bounds as follows. As pointed out above, the optimal solution of PKP is determined by a penalty value Π and a subset of items j with $\pi_j \leq \Pi$. Therefore, we consider sub–problem $PKP_j := PKP(\pi_j)$ for $j = 1,\ldots,n$. Recalling (4.6) each PKP_j is an instance of KP with item set $\{j, j+1,\ldots,n\}$ and capacity c where π_j is subtracted from the final solution value.

Fixing $\Pi = \pi_j$ for some j is only relevant for the final solution if item j is actually included in the solution. Hence, as in [15], we also consider sub–problem PKP_j^+, where item j is packed, a fixed penalty of $\Pi = \pi_j$ is subtracted from the objective function, and for the reminder of the solution a KP is solved with capacity $c - w_j$ and item set $\{j+1,\ldots,n\}$.

For both PKP_j resp. PKP_j^+ we consider the LP-relaxation as upper bound denoted by PKP_j^{LP} resp. PKP_j^{+LP}. It is easy to see that

$$z(PKP_j^+) \leq z(PKP_j) \tag{4.13}$$

$$z^* = \max_{j=1,\ldots,n} z(PKP_j) \leq \max_{j=1,\ldots,n} z(PKP_j^{LP}) =: UB_{sub} \tag{4.14}$$

$$z^* = \max_{j=1,\ldots,n} z(PKP_j^+) \leq \max_{j=1,\ldots,n} z(PKP_j^{+LP}) =: UB_{sub}^+ \tag{4.15}$$

The following dominance relations exist for the upper bounds UB_{sub}, UB_{sub}^+ and z^{LP}.

Proposition 5. *For any PKP instance, we have that*

$$UB_{sub}^+ \leq UB_{sub} \leq z^{LP} \tag{4.16}$$

and there are instances where the inequalities are strict.

Proof. Clearly, the restricted feasible domain of UB^{+}_{sub} cannot lead to a greater value than UB_{sub} and thus $UB^{+}_{sub} \leq UB_{sub}$. Let us denote by j' the item yielding UB_{sub}, i.e. $UB_{sub} = z(PKP^{LP}_{j'})$. Computing $PKP^{LP}(\Pi)$ with $\Pi = \pi_{j'}$ gives a feasible solution for the LP relaxation whose value is less than (or equal to) the optimal value z^{LP} but at least as large as UB_{sub}. The latter holds because *all* items are involved (and bounded according to (4.7)) in the computation while only items i with $\pi_i \leq \pi_{j'}$ are considered for solving $PKP^{LP}_{j'}$. This implies that $UB_{sub} \leq z^{LP}$.

To show that inequalities in (4.16) can be strict, consider the following PKP instance with $n = 2$ items, capacity $c = 7$ and the entries:

$$p_1 = 10, w_1 = 5, \pi_1 = 1; \quad p_2 = 6, w_2 = 4, \pi_2 = 2$$

For this instance we have $z^{LP} = 12$, $z(PKP^{LP}_1) = -1 + 10 = 9$, $z(PKP^{LP}_2) = -2 + 10 + \frac{2}{4}6 = 11$, $z(PKP^{+LP}_1) = -1 + 10 = 9$, $z(PKP^{+LP}_2) = -2 + 6 + \frac{3}{5}10 = 10$. Thus, we have:

$$UB^{+}_{sub} = 10 < UB_{sub} = 11 < z^{LP} = 12$$

□

Although the three bounds can be computed efficiently and can be expected to be reasonably close to the optimal value in practice, we show a negative result on their deviation from the optimum.

Proposition 6. *There are instances of PKP where the differences $(UB^{+}_{sub} - z^*)$, $(UB_{sub} - z^*)$ and $(z^{LP} - z^*)$ are arbitrarily large.*

Proof. Consider the following instance with $n = 2$ items, capacity $c = M$ and the following entries: $p_j = w_j = \frac{M}{2} + 1$ and $\pi_j = \frac{M}{2}$ for $j = 1, 2$. In an optimal solution only one item j is packed and, correspondingly, $z^* = p_j - \pi_j = 1$. Also, it is easy to see that $UB^{+}_{sub} = \frac{M}{2}$, which, in combination with (4.16), shows the claim.

□

Algorithmically, it is not difficult to see that all values $z(PKP_j^{LP})$ for $j = 1, \ldots, n$ can be computed in $O(n \log n)$ time. Also from a practical point of view, the effort hardly exceeds sorting. As a preprocessing step an auxiliary array is constructed containing all items sorted in decreasing order of efficiencies p_j/w_j. Then the problems PKP_j^{LP} are considered iteratively for $j = 1, \ldots, n$, i.e., in decreasing order of penalties π_j. First, PKP_1^{LP} is solved in linear time and the corresponding split item is identified. We keep a pointer to this split item in the sorted array of items. Moving to PKP_2^{LP}, we just remove item 1 from the solution and increase the split item, or possibly move to a new split item by shifting the pointer towards items with lower efficiency. All together, after sorting, all values $z(PKP_j^{LP})$ can be determined in linear time by one pass through the sorted array of items.

In [15], the authors presented an $O(n^2)$ procedure to compute all values $z(PKP_j^{+LP})$ for $j = 1, \ldots, n$. In the following, we show that in fact $O(n \log n)$ time is sufficient to perform this task.

Theorem 8. *All values $z(PKP_j^{+LP})$ for $j = 1, \ldots, n$ can be computed in $O(n \log n)$ time.*

Proof. The algorithm is based on an auxiliary data structure consisting of a binary tree. First, the items are sorted in decreasing order of efficiencies. Based on this sequence we construct a binary tree as follows: Each item corresponds to a leaf node of the tree. These are nodes at level 0. A parent node is associated with each pair of consecutive items (with a singleton remaining at the end for n odd) thus yielding other $\lceil \frac{n}{2} \rceil$ nodes in the level 1 of the tree. This process is iterated recursively up the tree, which trivially reaches a height of $O(\log n)$.

In each node v of the tree we store as $W(v)$ (resp. $P(v)$) the sum of weights (resp. profits) of all items corresponding to leaf nodes in the subtree rooted in v. Clearly, such a tree and its additional information can be built in $O(n)$ time.

For any given capacity c' the corresponding split item and also the value of the optimal LP-relaxation can be found in $O(\log n)$ time by starting at the root node and going down towards a leaf node by applying the following rule in every node v with left and right child nodes v^L and v^R:

If $W(v^L) > c'$ then set $v := v^L$.
Otherwise set $v := v^R$ and $c' := c' - W(v^L)$.

The item corresponding to the leaf node reached by this procedure is the split item. The solution value can be reported by keeping track of the $P(v)$ values during the pass through the tree.

In the main iteration of the algorithm we compute $z(PKP_j^{+LP})$ iteratively for $j = 1, \ldots, n$ in decreasing order of penalties π_j. First we remove item j permanently from consideration. This means that the leaf node corresponding to j is removed from the tree and all $O(\log n)$ labels $W(v)$ (resp. $P(v)$) on the unique path from this leaf to the root of the tree are updated by subtracting w_j (resp. p_j). Then we solve an LP-relaxation with capacity $c' := c - w_j$ and add $p_j - \pi_j$ to the objective function.

All together there are n iterations, each of which requiring $O(\log n)$ time to find the solution of the LP-relaxation and $O(\log n)$ time to update the labels of the binary tree.
Note that we might expect a considerable speed–up of the running time $O(n \log n)$ in a practical implementation since the tree looses vertices in each iteration and path contractions can be performed. □

4.3.3 Negative approximation result

The 0–1 Knapsack Problem admits basic approximation algorithms (see, e.g., [47]). PKP has "only" an additional penalty to consider in the objective with respect to KP. Thus, one might expect some straightforward approximation algorithm for this problem as well. Nonetheless, we prove here the general result that no polynomial time approximation algorithm exists for PKP (under $\mathcal{P} \neq \mathcal{NP}$).

Theorem 9. *PKP does not have a polynomial time approximation algorithm with a bounded approximation ratio unless $\mathcal{P} = \mathcal{NP}$.*

Proof. Similarly to the proof for KPS in Section 2.6.2, the theorem is proved by reduction from the Subset Sum Problem (SSP). Given n items j with integer

weights w'_j (with $j = 1, \ldots, n$) and a value W' (with $\sum_{j=1}^{n} w'_j > W'$), we recall that the decision version of SSP is an NP–complete problem and asks whether there exists a subset of items represented by x^* such that $\sum_{j=1}^{n} w'_j x^*_j = W'$.

We build an instance of PKP with n items, profits and weights $p_j = w_j = w'_j$, penalties $\pi_j = W' - 1$ (with $j = 1, \ldots, n$) and capacity $c = W'$. The capacity constraint implies that for every feasible solution there is $\sum_{j=1}^{n} p_j x_j = \sum_{j=1}^{n} w_j x_j \leq W'$. The penalty value will be equal to either $W' - 1$ if we pack at least one item or 0 otherwise, therefore the optimal solution of this PKP instance is bounded by $\sum_{j=1}^{n} p_j x_j - (W' - 1) \leq 1$. Not placing any item in the knapsack attains the trivial solution with value equal to 0. By integrality of the input data, this limits the optimal solution value to 0 or 1, where the latter value can be reached if and only if the Subset Sum Problem has a solution.

Hence, if there was a polynomial time approximation algorithm with a bounded approximation ratio, we could decide the Subset Sum Problem just by checking if the approximate solution of PKP is strictly positive. Clearly this is not possible unless $\mathcal{P} = \mathcal{NP}$. □

4.3.4 A basic dynamic programming algorithm

As recalled in [15], a straightforward pseudo–polynomial algorithm for PKP consists of solving j standard knapsack problems PKP_j^+ by the classical dynamic programming by weights running in $O(nc)$. The overall complexity is thus $O(n^2 c)$. The following proposition shows that the complexity can be reduced to $O(\max\{n \log n, nc\})$.

Proposition 7. *PKP can be solved with complexity* $O(\max\{n \log n, nc\})$.

Proof. It suffices to consider the items sorted by increasing penalty and to run the dynamic program for KP only once. If we denote by $F_j(d)$ the optimal solution value of the sub–problem of KP consisting of items $1, \ldots, j$ and capacity $d \leq c$, the optimal value of any PKP instance is simply given by

$$\max_{j=0,\ldots,n-1} \{F_j(c - w_{j+1}) + p_{j+1} - \pi_{j+1}\} \tag{4.17}$$

That is, we evaluate the choice of item $j+1$ as leading item just by considering the maximum profit reachable with the previous items in a knapsack with capacity $c-w_{j+1}$. The running time is in $O(\max\{n\log n, nc\})$, where the term $O(n\log n)$ is due to the sorting of the items by penalty. □

4.4 An exact solution approach

4.4.1 Overview

The DP algorithm of Proposition 7, hereafter denoted as DP_1, may be appealing whenever the capacity c is of reasonably limited size. As in KP, the recursion in the dynamic program computes an optimal solution by starting from the first item and by iteratively adding the other items. Nevertheless, as discussed in Section 1.5, the most effective algorithms for KP are based on the core problem. Our idea is to exploit the core concept for PKP similarly to the framework of the *Minknap* algorithm (see Section 1.5.2). We remark that the presence of penalties compromises in PKP the structure of an optimal solution with respect to a standard KP. This difference would typically affect the performance of an approach based on a core problem. Further, the presence of penalties limits the effectiveness of the classical dominance rule in KP based on the profits and the weights of the states. Anyhow, from a practical perspective it is still beneficial to run a dynamic programming algorithm starting from the split solution of KP and not from scratch. In addition, by narrowing the interval of penalty values which can possibly lead to an optimal solution, the "noise" added by the penalties can be further reduced.

We propose an exact approach involving two main steps. In the first step, we effectively compute an initial feasible solution for the problem and identify the relevant interval of penalties values possibly leading to an optimal solution. In the second step, we run a dynamic programming algorithm with states based on the core concept. In case the first step yields a reduced problem with a reasonably limited input size, we could as well launch the DP_1 algorithm. In the following subsections we describe the steps of the approach whose pseudo code is presented in Algorithm 5.

Algorithm 5 Exact solution approach

Input: PKP instance, parameters T_1, T_2, T_3, α.

▷ Step 1

$KP_1 =$ PKP without penalties;
$(\overline{z}, \overline{j}, \overline{f}) \leftarrow$ *ModMinknap*(KP_1);
Compute $z(PKP_j^{+LP})$ for $j = \overline{f}+1, \ldots, n$;
$UB = \max_j z(PKP_j^{+LP})$;
if $UB \leq \overline{z}$ **then** $z^* = \overline{z}$, $j^* = \overline{j}$; **return** (z^*, j^*); **end if**
$k = \arg\max_j z(PKP_j^{+LP})$;
$KP_2 = KP_1 \cap (x_j = 0 \; j = 1, \ldots, k-1)$;
$(\hat{z}, \hat{j}, \hat{f}) \leftarrow$ *ModMinknap*(KP_2);
if $\hat{z} > \overline{z}$ **then** $\overline{z} = \hat{z}$, $\overline{j} = \hat{j}$; **end if**
$l \leftarrow$ Apply (4.18);
$r \leftarrow$ Apply (4.19);
$\Pi_{max} = \pi_l$;
$\Pi_{min} = \pi_r$;
if $[\Pi_{min}, \Pi_{max}] = \emptyset$ **then** $z^* = \overline{z}$, $j^* = \overline{j}$; **return** (z^*, j^*); **end if**
$PKP' =$ PKP $\cap (x_j = 0 \; j = 1, \ldots, l-1; \; \Pi \geq \Pi_{min})$;
$n' = n - l + 1$;

▷ Step 2

if $n'c \leq T_1$ and $(r - l + 1) \geq T_2$ **then**
    $(z', j') \leftarrow DP_1(PKP')$;
    if $z' > \overline{z}$ **then** $z^* = z'$, $j^* = j'$; **else** $z^* = \overline{z}$, $j^* = \overline{j}$; **end if**
else
    $(z^*, j^*) \leftarrow DP_2(PKP', \overline{z}, \overline{j}, T_3, \alpha)$;
end if
return (z^*, j^*);

4.4.2 Step 1: Computing an initial feasible solution and the relevant interval of penalty values

The approach takes in input four parameters T_1, T_2, T_3, α and starts by solving the standard knapsack problem KP_1 given by disregarding the penalties of the items in PKP (lines 2-3 in Algorithm 5). This problem is solved as follows. Denote the index of the first item in the optimal solution of KP_1 (according to the ordering (4.6)) by $\overline{f}$. The corresponding first feasible solution of PKP has objective value $z(KP_1) - \pi_{\overline{f}}$.
Similarly to Proposition 2 in [15], the following proposition holds

Proposition 8. *All items $j = 1, \ldots, \overline{f} - 1$ can be discarded without loss of optimality.*

Proof. Since $z(KP_1)$ is the optimal solution value, including any item $j = 1, \ldots, \overline{f} - 1$ leads to a solution with profits less than (or equal to) $z(KP_1)$ and induces a penalty greater than (or equal to) $\pi_{\overline{f}}$. Consequently, every solution of PKP with at least one of these items included cannot improve the first solution value $z(KP_1) - \pi_{\overline{f}}$. □

Thus, if there is more than one optimal solution of KP_1, we are interested in the solution yielding the lowest penalty value for PKP (i.e. the highest index $\overline{f}$). This task is easily accomplished by considering a slight variant of *Minknap*, hereafter denoted as *ModMinknap*, which keeps track of all optimal solutions of KP_1 and the corresponding penalty values in PKP. In addition, we can compute PKP solutions during the iterations of *ModMinknap* just by tracking the largest penalty associated to each feasible state. We then take the overall best solution found for PKP. Denote by $\overline{z}$ its value and by $\overline{j}$ the index of the leading item.
We remark that *ModMinknap* is only a heuristic algorithm for PKP since it does not explicitly consider the penalties of the items in the iterations. At the same, it may "stumble" upon good quality solutions of PKP with just a negligible increase of the computational effort required for solving a KP instance.

After that, we compute $z(PKP_j^{+LP})$ for $j = \overline{f} + 1, \ldots, n$. If the maximum of these values is less that $\overline{z}$, we have already certified an optimal solution for PKP

(lines 4–6 in Algorithm 5). Otherwise we greedily consider the index k yielding the maximum $z(PKP_j^{+LP})$ and solve KP_1 without items $j = 1, \ldots, k-1$. We update the values of $\overline{z}$ and $\overline{j}$ if an improving solution is found (lines 7–10 in Algorithm 5).

Finally, we compare the values $z(PKP_j^{+LP})$ with the incumbent solution value $\overline{z}$ and narrow the range of possible penalty values that may lead to an optimal solution of PKP. More precisely, we define indexes l and r

$$l = \min \{j : z(PKP_j^{+LP}) > \overline{z}\}; \tag{4.18}$$

$$r = \max \{j : z(PKP_j^{+LP}) > \overline{z}\}. \tag{4.19}$$

The relevant interval of penalties is thus $[\Pi_{min}, \Pi_{max}]$, with $\Pi_{min} = \pi_r$ and $\Pi_{max} = \pi_l$ (line 11–14 in Algorithm 5). If this interval is empty, the current PKP solution is also optimal and the algorithm terminates. Otherwise we get a reduced PKP with only items $j = l, \ldots, n$ and the additional constraint on the penalty value $\Pi \geq \Pi_{min}$. Denote this problem by PKP' and its number of items by n', i.e. $n' = n - l + 1$ (lines 15–17 in Algorithm 5).

This first step is expected to be fast since it relies on solving two standard knapsack problems at most and on effectively computing upper bounds for sub–problems PKP_j^+. We remark that this step is also sufficient to compute an optimal solution for a large number of instances considered in the literature.

4.4.3 Step 2: A core–based dynamic programming algorithm

In this step we propose a core–based dynamic programming algorithm, hereafter denoted as DP_2, that constitutes a revisiting of *Minknap* algorithm. Notice that, if the size of the reduced problem PKP' is reasonably small and the number of relevant penalties is large, we could otherwise solve PKP' by DP_1 and take the best solution between $z(PKP')$ and $\overline{z}$. The choice between the algorithms is made by comparing the quantities $n'c$ and $(r - l + 1)$ with the threshold parameters T_1 and T_2 (lines 18–23 in Algorithm 5).

DP_2 algorithm searches in PKP' for better solutions than $\overline{z}$. In the following we describe the algorithm after some preliminary definitions. Given the sorting of the items $j = 1, \ldots, n'$ by decreasing $\frac{p_j}{w_j}$, we define an expanding core as the interval of items $C_{a,b} = \{a, \ldots, b\}$ with items a and b as variable extremes. Correspondingly, we define the set of 0–1 partial vectors enumerated within the core as

$$X_{a,b} = \{x_j \in \{0,1\}, j \in C_{a,b}\}. \tag{4.20}$$

Since in any iteration of the algorithm we will have the following situation

$$\overbrace{x_1, \ldots, x_{a-1}}^{x_j = 1}, C_{a,b}, \overbrace{x_{b+1}, \ldots, x_{n'}}^{x_j = 0} \tag{4.21}$$

we associate each partial vector $\tilde{x} \in X_{a,b}$ with a state $(\tilde{\nu}, \tilde{\mu}, \tilde{\pi}_{core}, \tilde{\pi}_{tot})$ where:

1. $\tilde{\nu} = \sum_{j=1}^{a-1} p_j + \sum_{j=a}^{b} p_j \tilde{x}_j$;
2. $\tilde{\mu} = \sum_{j=1}^{a-1} w_j + \sum_{j=a}^{b} w_j \tilde{x}_j$;
3. $\tilde{\pi}_{core} = \max_{j=a,\ldots,b} \{\pi_j : \tilde{x}_j = 1\}$;
4. $\tilde{\pi}_{tot} = \max\{\tilde{\pi}_{core}, \max_{j=1,\ldots,a-1} \pi_j\}$.

$\tilde{\nu}$ and $\tilde{\mu}$ are the profits and weights of a solution with variables in the core and all variables to the left of the core; $\tilde{\pi}_{core}$ represents the maximum penalty of the items selected in the core while $\tilde{\pi}_{tot}$ is the overall maximum penalty of the state. Each state with $\tilde{\mu} \leq c$ and $\tilde{\pi}_{tot} \geq \Pi_{min}$ represents a feasible solution of PKP' with value $\tilde{\nu} - \tilde{\pi}_{tot}$.
We can now sketch the main steps of DP_2 in the following pseudo code.

Algorithm 6 $DP_2(PKP', \overline{z}, \overline{j}, T_3, \alpha)$

Sort items in PKP' by decreasing $\frac{p_j}{w_j}$;
$KP' = PKP'$ without penalties;
Find the split item s' of KP';
$a = b = s'$, $C_{a,b} = \{s'\}$; $X_{a,b} = \{(0),(1)\}$;
Reduce set $X_{a,b}$;
while $X_{a,b} \neq \emptyset$ and $(b - a + 1 < n')$ **do**
    $a \leftarrow a - 1$;
    if $u_0^a > \overline{z}$ **then**
        if $\tilde{u}^a > \overline{z}$ **then**
            $X_{a,b} \leftarrow Merge(a, X_{a+1,b}, \Pi_{min}, T_3, \alpha)$;
            Update $(\overline{z}, \overline{j})$;
            Reduce set $X_{a,b}$;
        end if
    end if
    $b \leftarrow b + 1$;,
    if $u_1^b > \overline{z}$ **then**
        if $\tilde{u}^b > \overline{z}$ **then**
            $X_{a,b} \leftarrow Merge(b, X_{a,b-1}, \Pi_{min}, T_3, \alpha)$;
            Update $(\overline{z}, \overline{j})$;
            Reduce set $X_{a,b}$;
        end if
    end if
end while
return $(\overline{z}, \overline{j})$;

The algorithm takes in input PKP', the current solution $(\overline{z}, \overline{j})$ and parameters T_3, α.

We first sort the items j $(j = 1, \ldots, n')$ by decreasing $\frac{p_j}{w_j}$ and find the split item s' of the standard knapsack problem (KP') induced by disregarding the penalties in PKP'. We then initialize the core with item s' only (lines 1–4 of the pseudo code).

Then, we enlarge the core as in *Minknap* (while–loop in lines 6–23) by alternately evaluating the removal of an item a from the left (lines 7–14) and the insertion of an item b from the right (lines 15–22). The expansion of the core is performed by a dynamic programming with states through a procedure, denoted as *Merge*, which iteratively yields undominated states in the enlarged set $X_{a,b} = X_{a+1,b} + a$ or $X_{a,b} = X_{a,b-1} + b$. We may update the current solution $(\overline{z}, \overline{j})$ if an improved solution is found while enumerating the core (lines 11 and 19).

The dynamic programming with states is combined with an upper bound test to reduce the number of states (lines 5, 12 and 20) and two upper bound tests to limit the insertion of the variables in the core (lines 8–9 and 16–17). The algorithm terminates whenever either the number of states is 0 or all variables have been enumerated in the core.

The ingredients of the algorithm are detailed in the following.

Dynamic programming with states

The *Merge* procedure performs the enumeration of the variables in the core by resembling the procedure introduced in [74] and used in *Minknap*, which in turn corresponds to the recursions of the *primal–dual* dynamic programming algorithm in [78] (see Section 1.3.3).

The proposed procedure merges, in any iteration, the current set of states X and $X + d$, where $X + d$ is set X with profits, weights and penalties of the states updated according to the removal/insertion of item d from/in the knapsack. In the merging operation the states are kept ordered by increasing weights so as to effectively apply a dominance rule for PKP. The classical dominance rule in KP considers the weights and profits of the states (see Section 1.3.2). For PKP, let us define the quantity $\rho = \nu - \max\{\pi_{core}, \Pi_{min}\}$ which represents

the difference between the profit of a state and the minimum penalty that the state must have for yielding an optimal solution. This penalty corresponds to the maximum between Π_{min} and π_{core} since, due to the enumeration of the core, for any state π_{core} constitutes a minimum penalty value in all states originating from it while Π_{min} is the minimum penalty required in any solution with a value greater than $\overline{z}$. We introduce the following dominance rule for two generic states i and j.

Proposition 9. *Given states i and j and their quantities fulfilling*

$$\mu^i \leq \mu^j, \qquad \nu^i \geq \nu^j, \qquad \rho^i \geq \rho^j, \tag{4.22}$$

Then state j is said to be dominated by state i and can be discarded in the search for an optimal solution of PKP.

Proof. The first two conditions represent the dominance of state i in the standard KP. The condition $\rho^i \geq \rho^j$ implies that all successive states deriving from state i and possibly optimal for PKP (i.e. with a penalty greater than Π_{min}) would have a no worse solution value than those deriving from state j. □

We remark that, given the presence of penalties, the ordering of states by increasing weights does not imply the ordering of the profits as in *Minknap*. To better detect situations of dominance, we apply the rule in Proposition 9 by comparing each state with a number of states (with a lower weight) given by parameter α.

Whenever only the condition involving the penalties prevents the fathoming of state j, we may combine the dominance rule with an upper bound on state j depending on a penalty value $\pi > \max\{\pi^j_{core}, \Pi_{min}\}$.
This upper bound, denoted by $UB(\pi)^j$, is computed as follows. We first solve the linear relaxation of the KP induced by packing the items selected in the core for state j and by disregarding the items outside the core with a higher penalty than π. From the optimal solution value of this problem we then subtract the maximum value between π^j_{core} and Π_{min}. The following proposition holds

Proposition 10. *Given two states i and j and the quantities*

$$\mu^i \leq \mu^j, \qquad \nu^i \geq \nu^j, \qquad \rho^i < \rho^j, \tag{4.23}$$

consider the maximum penalty $\hat{\pi}$ which would not induce the dominance of state i according to (4.22), i.e. $\hat{\pi} = \max\{\pi : \nu^j - \pi > \rho^i\}$. If $UB(\hat{\pi})^j \leq \overline{z}$, then state j can be discarded.

Proof. We analyze the solution values deriving from state j when the overall maximum penalty is upper bounded by a quantity π'. For any $\pi' \leq \hat{\pi}$, since $UB(\hat{\pi})^j \leq \overline{z}$ we can discard state j because all states deriving from state j cannot reach a solution values greater than $\overline{z}$. Likewise, we can as well discard state j if $\pi' > \hat{\pi}$ since this condition would induce a dominance of state i. □

Computing $UB(\pi)$ has complexity $O(n)$ and would be time–consuming if the number of states involved is sufficiently large. Thus, we calculate this bound only if the number of states overcomes the threshold value T_3.

As in *Minknap*, in any iteration the running time of the *Merge* procedure is linear with the number of states. In each iteration the number of states can be bounded by $O(n'c)$ since the weights of the states range from 1 to $2c$ and there could be at most n' (with $n' \leq n$) states with the same weight due to different values of ρ. Hence, the running time of DP_2 is bounded by $O(2^n)$ for the enumeration of the states in the core in combination with the pseudopolynomial bound $O(n^2c)$.

Reduction of the states

To further reduce the set of states, we also perform an upper bound test in constant time for each state. In any iteration, we compute the following upper bound for a state i associated with a vector in $X_{a,b}$:

$$UB^i = \begin{cases} \rho^i + (c - \mu^i)\frac{p_{b+1}}{w_{b+1}} & \text{if } \mu^i \leq c \\ \rho^i + (c - \mu^i)\frac{p_{a-1}}{w_{a-1}} & \text{if } \mu^i > c \end{cases} \tag{4.24}$$

and discard state i if $UB^i \leq \overline{z}$. These upper bounds are computed by replacing the integrality constraint on x_{a-1} and x_{b+1} with $x_{a-1} \geq 0$ and $x_{b+1} \geq 0$ and by disregarding the penalty values of the variables outside the core.

Upper bound tests on the variables outside the core

Since the insertion of variables in the core may be computationally expensive, we perform two upper bound tests whenever an item j is candidate to be included in the core.
We first compute similar bounds to the ones proposed in [27] for KP (i.e bounds (1.16) and (1.17) in Section 1.2.6). Let us denote by u_0^j an upper bound on PKP' without item j. Also, let us denote by u_1^j the upper bound when item j is packed. The following bounds are computed in constant time for each item j:

$$u_0^j = p' - p_j - \Pi_{min} + (c - w' + w_j)\frac{p_{s'}}{w_{s'}} \quad j = 1, \ldots, s'-1 \tag{4.25}$$

$$u_1^j = p' + p_j - \max\{\pi_j, \Pi_{min}\} + (c - w' - w_j)\frac{p_{s'}}{w_{s'}} \quad j = s'+1, \ldots, n \tag{4.26}$$

Here $w' = \sum_{j=1}^{s'-1} w_j$ and $p' = \sum_{j=1}^{s'-1} p_j$ respectively represent the weight and the profit of the split solution of KP'. If u_0^j (resp. u_1^j) $\leq \overline{z}$, we can fix variable $x_j = 1$ (resp. $x_j = 0$).

In cascade, we may perform a second test by computing a stronger upper bound in linear time with the number of states. As in *Minknap*, we evaluate the impacts of removing (inserting) item j with $j < s'$ ($j > s'$) in all states in the current set X, namely we derive states $i \in X + j$ and compute upper bounds (4.24) on these states. A valid upper bound for item j, denoted as $\tilde{u}^j$, is constituted by the maximum of these bounds. As pointed out in [76], $\tilde{u}^j$ can be seen as a generalization of the enumerative bound in [60] (bound (1.13) in Section 1.2.5). If $\tilde{u}^j \leq \overline{z}$, then variable x_j is fixed to the value taken in the split solution.

After this second step, the optimal solution value z^* and the optimal leading item j^* are returned. The optimal solution set of items can be determined by solving the standard knapsack problem $PKP_{j^*}^+$.

4.5 Computational results

All tests were performed on an Intel i7 CPU @ 2.4 GHz with 8 GB of RAM. The code was implemented in the C++ programming language. We generated the instances according to the generation scheme proposed in [15][2]. We considered 2 types of weights: $a1$ and $a2$. In the former type, the weights are randomly distributed in $[1, R]$, with R being an arbitrary parameter. In the latter type, the weights are equal to $\frac{R}{2} + v$, with v uniformly distributed in $[0, \frac{R}{2}]$. Basically small weights are not considered in $a2$.
We generated 8 classes of penalties $(\pi 1, \ldots, \pi 8)$ and 7 classes of profits $(p1, \ldots, p7)$ according to different correlations of penalties/profits with the weights, as illustrated in Table 4.1. The first 6 correlations correspond to classical correlations in KP instances. In class $\pi 7$ penalties π_j are equal to $R - w_j + 1$ (constant perimeter correlation) while in class $\pi 8$ we set $\pi_j = \frac{R}{w_j}$ (constant area correlation). In class $p7$ we set $p_j = \pi_j w_j$. Finally, three different values of the ratio τ between the knapsack capacity and the sum of the weights of the items are considered: 0.5, 0.1 and 0.01.

π type	Correlation	p type
$\pi 1$	No correlation	$p1$
$\pi 2$	Weak correlation	$p2$
$\pi 3$	Strong correlation	$p3$
$\pi 4$	Inverse strong correlation	$p4$
$\pi 5$	Almost strong correlation	$p5$
$\pi 6$	Subset-sum correlation	$p6$
$\pi 7$	Constant perimeter	
$\pi 8$	Constant area	
	Profit = area	$p7$

Table 4.1 Correlation types from [15].

We first generated instances with 1000 items and $R = 1000$. Within each category, 5 instances were tested for a total of 1680 instances. Similarly, we generated 1680 instances with 10000 items and $R = 10000$. We compared the solutions reached by our approach, the exact approach in [15] and CPLEX 12.5 running on model (PKP). After a preliminary experimentation, we considered

[2]We would like to thank Alberto Ceselli and Giovanni Righini for providing us with the code of their algorithm and the generation scheme of the instances.

the following values of the parameters for our approach: $\alpha = 15$, $T_1 = 5 * 10^9$, $T_2 = \frac{n}{10}$, $T_3 = 3 * 10^6$. The parameters of the IP solver were set to their default values.

The results are summarized in Tables 4.2 and 4.3 in terms of average, maximum CPU time and number of optima obtained within a time limit of 100 seconds. The average CPU times consider also the cases where the time limit is reached. The results are aggregated by profit classes and weight types. Each entry in the tables reports the results over 120 instances. The detailed results for all correlations and capacity ratios are reported in Tables 4.4–4.7.

$n = 1000$		CPLEX 12.5			Algorithm in [15]			Exact approach		
Profit class	Weight type	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
$p1$	$a1$	0.18	0.25	120	0.00	0.01	120	0.00	0.00	120
	$a2$	0.19	0.48	120	0.00	0.01	120	0.00	0.03	120
$p2$	$a1$	0.39	1.69	120	0.00	0.01	120	0.00	0.10	120
	$a2$	1.12	6.32	120	0.00	0.01	120	0.01	0.27	120
$p3$	$a1$	3.90	100.00	117	0.04	0.89	120	0.01	0.40	120
	$a2$	6.83	100.00	117	0.50	8.02	120	0.02	0.29	120
$p4$	$a1$	59.56	100.00	57	0.10	1.38	120	0.02	0.18	120
	$a2$	66.97	100.00	46	0.28	7.75	120	0.04	0.26	120
$p5$	$a1$	4.13	100.00	117	0.03	0.89	120	0.02	0.20	120
	$a2$	14.97	100.00	114	0.41	4.50	120	0.07	1.40	120
$p6$	$a1$	2.67	90.38	120	0.00	0.06	120	0.00	0.03	120
	$a2$	2.97	13.57	120	0.01	0.15	120	0.01	0.08	120
$p7$	$a1$	27.58	100.00	89	0.00	0.01	120	0.00	0.02	120
	$a2$	38.24	100.00	75	0.01	0.06	120	0.01	0.07	120

Table 4.2 Summary results for instances with 1000 items and different correlations between profits and weights: time (s) and number of optima over 120 instances.

$n = 10000$		CPLEX 12.5			Algorithm in [15]			Exact approach		
Profit class	Weight type	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
$p1$	$a1$	0.95	2.68	120	0.01	0.02	120	0.01	0.03	120
	$a2$	4.19	100.00	117	0.01	0.04	120	0.01	0.03	120
$p2$	$a1$	5.41	33.84	120	0.01	0.02	120	0.02	0.13	120
	$a2$	12.43	100.00	114	0.01	0.07	120	0.04	0.73	120
$p3$	$a1$	44.61	100.00	74	25.59	100.00	96	2.59	58.46	120
	$a2$	74.13	100.00	46	48.26	100.00	74	5.53	29.00	120
$p4$	$a1$	91.60	100.00	11	17.68	100.00	106	2.81	18.88	120
	$a2$	94.69	100.00	7	26.34	100.00	106	7.82	80.02	120
$p5$	$a1$	25.45	100.00	95	10.70	100.00	113	2.66	70.71	120
	$a2$	65.99	100.00	48	23.74	100.00	101	7.04	58.60	120
$p6$	$a1$	83.08	100.00	58	0.67	40.17	120	0.22	5.38	120
	$a2$	81.05	100.00	44	3.62	100.00	119	1.95	16.65	120
$p7$	$a1$	51.00	100.00	63	0.17	0.94	120	0.42	2.50	120
	$a2$	40.12	100.00	75	0.54	3.94	120	1.41	11.46	120

Table 4.3 Summary results for instances with 10000 items and different correlations between profits and weights: time (s) and number of optima over 120 instances.

$n = 1000$			CPLEX 12.5			Algorithm in [15]			Exact approach		
Weight type	τ	Profit class	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
$a1$	0.5	$p1$	0.19	0.25	40	0.00	0.00	40	0.00	0.00	40
		$p2$	0.32	0.92	40	0.00	0.01	40	0.00	0.10	40
		$p3$	10.59	100.00	37	0.10	0.89	40	0.03	0.40	40
		$p4$	62.27	100.00	16	0.14	1.34	40	0.02	0.05	40
		$p5$	11.08	100.00	37	0.09	0.89	40	0.04	0.20	40
		$p6$	2.29	18.94	40	0.00	0.00	40	0.00	0.02	40
		$p7$	36.28	100.00	26	0.00	0.01	40	0.00	0.01	40
	0.1	$p1$	0.20	0.25	40	0.00	0.01	40	0.00	0.00	40
		$p2$	0.49	1.13	40	0.00	0.00	40	0.00	0.05	40
		$p3$	0.54	1.44	40	0.02	0.11	40	0.01	0.04	40
		$p4$	74.13	100.00	11	0.12	1.38	40	0.02	0.18	40
		$p5$	0.93	10.24	40	0.01	0.06	40	0.01	0.05	40
		$p6$	4.23	90.38	40	0.00	0.01	40	0.00	0.03	40
		$p7$	28.61	100.00	29	0.00	0.01	40	0.01	0.02	40
	0.01	$p1$	0.14	0.18	40	0.00	0.01	40	0.00	0.00	40
		$p2$	0.35	1.69	40	0.00	0.00	40	0.00	0.01	40
		$p3$	0.58	8.34	40	0.00	0.03	40	0.00	0.01	40
		$p4$	42.27	100.00	30	0.04	0.20	40	0.01	0.03	40
		$p5$	0.38	3.65	40	0.00	0.01	40	0.00	0.01	40
		$p6$	1.48	4.05	40	0.00	0.06	40	0.00	0.01	40
		$p7$	17.87	100.00	34	0.00	0.01	40	0.00	0.02	40
$a2$	0.5	$p1$	0.18	0.32	40	0.00	0.00	40	0.00	0.00	40
		$p2$	0.47	3.19	40	0.00	0.01	40	0.00	0.00	40
		$p3$	10.66	100.00	37	0.89	8.02	40	0.03	0.11	40
		$p4$	53.29	100.00	19	0.50	7.75	40	0.04	0.11	40
		$p5$	23.32	100.00	36	0.65	4.50	40	0.12	1.40	40
		$p6$	4.65	13.57	40	0.00	0.01	40	0.02	0.08	40
		$p7$	17.76	100.00	33	0.00	0.01	40	0.01	0.03	40
	0.1	$p1$	0.22	0.33	40	0.00	0.00	40	0.00	0.00	40
		$p2$	1.52	6.32	40	0.00	0.01	40	0.01	0.27	40
		$p3$	3.40	41.67	40	0.41	2.17	40	0.03	0.29	40
		$p4$	87.51	100.00	6	0.31	7.30	40	0.04	0.26	40
		$p5$	10.75	100.00	38	0.38	1.63	40	0.07	0.38	40
		$p6$	1.91	5.30	40	0.00	0.08	40	0.01	0.04	40
		$p7$	35.62	100.00	26	0.01	0.03	40	0.01	0.07	40
	0.01	$p1$	0.18	0.48	40	0.00	0.01	40	0.00	0.03	40
		$p2$	1.38	4.26	40	0.00	0.01	40	0.00	0.01	40
		$p3$	6.43	18.53	40	0.20	1.17	40	0.02	0.03	40
		$p4$	60.12	100.00	21	0.04	0.14	40	0.03	0.05	40
		$p5$	10.84	60.54	40	0.19	0.84	40	0.02	0.04	40
		$p6$	2.36	6.48	40	0.03	0.15	40	0.01	0.07	40
		$p7$	61.34	100.00	16	0.01	0.06	40	0.02	0.06	40

Table 4.4 Detailed results for instances with 1000 items and different correlations between profits and weights: time (s) and number of optima over 40 instances.

$n = 10000$			CPLEX 12.5			Algorithm in [15]			Exact approach		
Weight type	τ	Profit class	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
$a1$	0.5	$p1$	1.00	2.28	40	0.01	0.02	40	0.02	0.03	40
		$p2$	3.73	9.40	40	0.01	0.02	40	0.02	0.03	40
		$p3$	62.62	100.00	16	45.29	100.00	23	6.65	58.46	40
		$p4$	82.03	100.00	8	17.43	100.00	34	2.79	10.73	40
		$p5$	53.44	100.00	20	24.99	100.00	33	7.16	70.71	40
		$p6$	90.65	100.00	12	0.01	0.06	40	0.23	2.56	40
		$p7$	21.31	100.00	33	0.09	0.33	40	0.15	0.51	40
	0.1	$p1$	1.12	2.68	40	0.01	0.01	40	0.01	0.02	40
		$p2$	11.05	33.84	40	0.01	0.02	40	0.02	0.13	40
		$p3$	55.29	100.00	21	26.62	100.00	33	1.02	8.73	40
		$p4$	95.24	100.00	2	16.64	100.00	37	3.54	18.88	40
		$p5$	20.70	100.00	35	6.24	78.90	40	0.76	6.13	40
		$p6$	77.25	100.00	24	0.09	1.97	40	0.33	5.38	40
		$p7$	64.61	100.00	15	0.27	0.94	40	0.57	2.26	40
	0.01	$p1$	0.73	1.28	40	0.00	0.01	40	0.00	0.01	40
		$p2$	1.45	2.74	40	0.00	0.02	40	0.01	0.02	40
		$p3$	15.92	100.00	37	4.86	59.95	40	0.12	1.00	40
		$p4$	97.53	100.00	1	18.97	100.00	35	2.10	14.20	40
		$p5$	2.22	6.56	40	0.86	7.79	40	0.07	0.56	40
		$p6$	81.35	100.00	22	1.91	40.17	40	0.11	1.69	40
		$p7$	67.09	100.00	15	0.14	0.56	40	0.54	2.50	40
$a2$	0.5	$p1$	1.93	11.22	40	0.01	0.02	40	0.02	0.02	40
		$p2$	3.86	11.72	40	0.01	0.02	40	0.02	0.03	40
		$p3$	70.77	100.00	18	53.23	100.00	23	8.09	25.55	40
		$p4$	86.05	100.00	6	24.86	100.00	35	8.72	26.06	40
		$p5$	60.26	100.00	19	24.30	100.00	32	9.93	58.60	40
		$p6$	86.29	100.00	7	0.08	0.95	40	2.62	16.65	40
		$p7$	14.30	100.00	36	0.31	2.23	40	0.48	3.34	40
	0.1	$p1$	1.27	2.19	40	0.01	0.02	40	0.01	0.02	40
		$p2$	10.99	29.04	40	0.01	0.02	40	0.04	0.66	40
		$p3$	66.31	100.00	19	50.95	100.00	24	4.33	14.29	40
		$p4$	98.01	100.00	1	34.25	100.00	35	11.28	80.02	40
		$p5$	54.25	100.00	21	27.97	100.00	35	5.97	40.59	40
		$p6$	82.95	100.00	13	0.29	1.47	40	2.28	7.87	40
		$p7$	51.33	100.00	20	0.77	3.94	40	1.83	11.46	40
	0.01	$p1$	9.37	100.00	37	0.01	0.04	40	0.01	0.03	40
		$p2$	22.45	100.00	34	0.01	0.07	40	0.06	0.73	40
		$p3$	85.32	100.00	9	40.59	100.00	27	4.18	29.00	40
		$p4$	100.00	100.00	0	19.92	100.00	36	3.46	15.36	40
		$p5$	83.47	100.00	8	18.96	100.00	34	5.22	20.51	40
		$p6$	73.90	100.00	24	10.50	100.00	39	0.96	3.90	40
		$p7$	54.74	100.00	19	0.55	3.47	40	1.92	8.74	40

Table 4.5 Detailed results for instances with 10000 items and different correlations between profits and weights: time (s) and number of optima over 40 instances.

$n = 1000$			CPLEX 12.5			Algorithm in [15]			Exact approach		
Weight type	τ	Penalty class	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
$a1$	0.5	$\pi 1$	6.65	100.00	33	0.01	0.05	35	0.01	0.03	35
		$\pi 2$	16.33	100.00	30	0.02	0.14	35	0.01	0.12	35
		$\pi 3$	19.56	100.00	29	0.08	0.89	35	0.02	0.14	35
		$\pi 4$	21.98	100.00	28	0.03	0.39	35	0.01	0.06	35
		$\pi 5$	22.95	100.00	28	0.05	0.38	35	0.01	0.10	35
		$\pi 6$	25.89	100.00	27	0.04	0.89	35	0.01	0.05	35
		$\pi 7$	15.49	100.00	30	0.09	1.34	35	0.01	0.03	35
		$\pi 8$	11.73	100.00	31	0.06	0.63	35	0.03	0.40	35
	0.1	$\pi 1$	7.53	100.00	33	0.00	0.05	35	0.01	0.18	35
		$\pi 2$	15.01	100.00	30	0.01	0.08	35	0.01	0.08	35
		$\pi 3$	29.18	100.00	25	0.01	0.08	35	0.01	0.03	35
		$\pi 4$	17.96	100.00	29	0.01	0.06	35	0.01	0.03	35
		$\pi 5$	18.44	100.00	29	0.01	0.09	35	0.01	0.03	35
		$\pi 6$	12.68	100.00	31	0.01	0.11	35	0.00	0.04	35
		$\pi 7$	17.93	100.00	30	0.09	1.38	35	0.01	0.05	35
		$\pi 8$	5.98	100.00	33	0.03	0.58	35	0.01	0.05	35
	0.01	$\pi 1$	0.70	3.83	35	0.00	0.00	35	0.00	0.01	35
		$\pi 2$	7.19	100.00	33	0.00	0.03	35	0.01	0.03	35
		$\pi 3$	7.52	100.00	34	0.00	0.05	35	0.00	0.03	35
		$\pi 4$	20.09	100.00	30	0.01	0.05	35	0.01	0.03	35
		$\pi 5$	16.57	100.00	32	0.01	0.05	35	0.00	0.03	35
		$\pi 6$	3.37	30.16	35	0.00	0.01	35	0.00	0.01	35
		$\pi 7$	15.94	100.00	30	0.02	0.20	35	0.00	0.01	35
		$\pi 8$	0.70	8.34	35	0.01	0.09	35	0.00	0.01	35
$a2$	0.5	$\pi 1$	10.77	100.00	32	0.02	0.14	35	0.02	0.17	35
		$\pi 2$	16.60	100.00	31	0.08	1.07	35	0.07	1.40	35
		$\pi 3$	22.18	100.00	29	0.42	4.57	35	0.04	0.32	35
		$\pi 4$	19.38	100.00	30	0.51	7.81	35	0.03	0.21	35
		$\pi 5$	21.57	100.00	29	0.28	3.00	35	0.03	0.28	35
		$\pi 6$	11.32	100.00	32	0.47	8.02	35	0.03	0.23	35
		$\pi 7$	9.68	100.00	32	0.49	7.75	35	0.02	0.11	35
		$\pi 8$	14.59	100.00	30	0.05	0.80	35	0.01	0.09	35
	0.1	$\pi 1$	14.78	100.00	32	0.02	0.17	35	0.05	0.38	35
		$\pi 2$	21.28	100.00	28	0.04	0.30	35	0.02	0.22	35
		$\pi 3$	18.82	100.00	29	0.22	2.16	35	0.03	0.27	35
		$\pi 4$	30.67	100.00	25	0.28	2.17	35	0.03	0.13	35
		$\pi 5$	25.16	100.00	27	0.22	1.98	35	0.02	0.12	35
		$\pi 6$	28.61	100.00	26	0.19	1.63	35	0.02	0.09	35
		$\pi 7$	10.08	100.00	32	0.26	7.30	35	0.02	0.11	35
		$\pi 8$	11.68	100.00	31	0.04	0.44	35	0.01	0.07	35
	0.01	$\pi 1$	1.60	12.74	35	0.00	0.01	35	0.01	0.03	35
		$\pi 2$	28.11	100.00	28	0.01	0.14	35	0.01	0.04	35
		$\pi 3$	26.29	100.00	28	0.15	0.73	35	0.02	0.05	35
		$\pi 4$	19.68	100.00	31	0.11	1.17	35	0.02	0.05	35
		$\pi 5$	33.97	100.00	25	0.10	0.67	35	0.02	0.05	35
		$\pi 6$	24.26	100.00	29	0.12	0.84	35	0.02	0.06	35
		$\pi 7$	17.34	100.00	30	0.02	0.10	35	0.01	0.03	35
		$\pi 8$	11.77	100.00	31	0.02	0.16	35	0.00	0.02	35

Table 4.6 Detailed results for instances with 1000 items and different correlations between penalties and weights: time (s) and number of optima over 35 instances.

$n = 10000$			CPLEX 12.5			Algorithm in [15]			Exact approach		
Weight type	τ	Penalty class	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt	Average time (s)	Max time (s)	#Opt
$a1$	0.5	$\pi 1$	26.96	100.00	29	1.07	8.74	35	1.04	10.54	35
		$\pi 2$	51.68	100.00	18	4.93	100.00	34	0.89	8.74	35
		$\pi 3$	45.50	100.00	20	8.83	100.00	33	1.06	5.78	35
		$\pi 4$	52.72	100.00	17	24.28	100.00	28	2.22	17.67	35
		$\pi 5$	54.65	100.00	17	18.01	100.00	29	1.66	10.09	35
		$\pi 6$	55.55	100.00	16	23.99	100.00	27	2.46	20.65	35
		$\pi 7$	38.21	100.00	25	11.72	100.00	31	1.08	10.73	35
		$\pi 8$	34.49	100.00	27	7.54	100.00	33	9.05	70.71	35
	0.1	$\pi 1$	27.19	100.00	28	0.88	18.70	35	0.72	18.88	35
		$\pi 2$	48.67	100.00	22	2.17	20.73	35	0.70	6.27	35
		$\pi 3$	54.65	100.00	18	7.52	100.00	33	1.30	18.19	35
		$\pi 4$	59.08	100.00	17	15.20	100.00	32	0.91	7.16	35
		$\pi 5$	57.49	100.00	19	8.46	100.00	34	1.00	8.53	35
		$\pi 6$	51.93	100.00	21	7.55	100.00	34	0.39	2.26	35
		$\pi 7$	48.77	100.00	22	11.15	100.00	32	0.85	8.59	35
		$\pi 8$	23.95	100.00	30	4.07	73.67	35	1.27	8.73	35
	0.01	$\pi 1$	21.36	100.00	30	0.17	3.00	35	0.31	6.57	35
		$\pi 2$	37.31	100.00	26	6.20	73.05	35	0.58	5.66	35
		$\pi 3$	45.55	100.00	21	2.26	33.15	35	0.28	3.69	35
		$\pi 4$	45.75	100.00	21	2.36	59.95	35	0.33	2.50	35
		$\pi 5$	49.01	100.00	21	1.25	10.15	35	0.46	5.62	35
		$\pi 6$	44.44	100.00	23	1.28	26.15	35	0.58	14.20	35
		$\pi 7$	38.94	100.00	22	13.75	100.00	31	0.47	3.36	35
		$\pi 8$	21.96	100.00	31	3.30	100.00	34	0.35	6.39	35
$a2$	0.5	$\pi 1$	50.50	100.00	18	4.70	44.55	35	2.91	17.61	35
		$\pi 2$	51.30	100.00	18	16.08	100.00	30	4.55	53.64	35
		$\pi 3$	46.81	100.00	21	19.17	100.00	29	5.14	42.97	35
		$\pi 4$	55.72	100.00	19	14.48	100.00	32	4.80	58.60	35
		$\pi 5$	52.02	100.00	19	20.40	100.00	30	4.84	29.21	35
		$\pi 6$	47.85	100.00	21	18.55	100.00	29	4.29	39.81	35
		$\pi 7$	37.68	100.00	24	11.97	100.00	32	3.61	25.55	35
		$\pi 8$	27.78	100.00	26	12.14	100.00	33	4.02	26.06	35
	0.1	$\pi 1$	46.84	100.00	21	11.89	100.00	33	3.47	41.73	35
		$\pi 2$	49.39	100.00	21	15.23	100.00	34	3.55	40.59	35
		$\pi 3$	57.48	100.00	18	21.44	100.00	29	4.78	28.76	35
		$\pi 4$	60.68	100.00	18	19.71	100.00	31	3.62	19.90	35
		$\pi 5$	73.48	100.00	10	23.79	100.00	29	4.05	14.26	35
		$\pi 6$	57.68	100.00	18	23.00	100.00	31	6.56	80.02	35
		$\pi 7$	41.52	100.00	22	8.76	100.00	33	1.70	7.87	35
		$\pi 8$	30.19	100.00	26	6.75	100.00	34	1.70	15.50	35
	0.01	$\pi 1$	61.19	100.00	16	2.94	26.59	35	3.98	29.00	35
		$\pi 2$	60.12	100.00	19	5.40	49.36	35	3.37	20.51	35
		$\pi 3$	65.69	100.00	14	15.40	100.00	31	1.87	12.09	35
		$\pi 4$	77.25	100.00	10	16.77	100.00	30	2.39	13.24	35
		$\pi 5$	73.10	100.00	12	17.29	100.00	30	2.08	11.25	35
		$\pi 6$	60.01	100.00	18	16.08	100.00	30	1.71	8.97	35
		$\pi 7$	58.23	100.00	17	24.62	100.00	30	1.70	6.82	35
		$\pi 8$	34.99	100.00	25	4.95	51.47	35	0.97	6.53	35

Table 4.7 Detailed results for instances with 10000 items and different correlations between penalties and weights: time (s) and number of optima over 35 instances.

From Tables 4.2 and 4.3 we see that, for the instances with 1000 items, both the proposed exact approach and the algorithm in [15] outperform CPLEX 12.5 which does not reach all the optima within the time limit. Although the performances of the algorithms are similar, we note that our approach generally performs slightly better and requires 1.4 seconds at most for solving to optimality all instances.

In the largest instances with 10000 items, our algorithm strongly outperforms both CPLEX 12.5 and the algorithm in [15]. Our approach is capable of reaching all optima with limited CPU time (80 seconds at most for an instance in class $p4$) while the solver and the competing algorithm run out of time for several large instances. The largest differences in computational times involve instances in classes $p3$, $p4$ and $p5$.

The most challenging instances for our algorithm turned out to be the ones without small weights ($a2$). In general, the absence of small weights might increase the computational effort required for solving even standard KP instances (as pointed out, e.g., in [15]) and this is presumably the reason of the increase in CPU times of our algorithm as well.

In many instances, the first main step relying on solving standard KPs is sufficient to certificate an optimal solution for PKP. Indeed, this constitutes a remarkable strength of our procedures. In Tables 4.8 and 4.9 we report the percentage of the optimal solutions already computed by the first step of the procedure for the instances with 1000 and 10000 items respectively. Averaged computational times (% of the total CPU time) of the two steps of our approach are also reported. Finally, we report the average and maximum values (in thousands) of the maximum number of states reached by DP_2 algorithm in each instance. We point out that DP_1 algorithm is called a limited number of times with respect to DP_2 (5% of the cases) and mainly in the smallest instances with 1000 items.

Exact approach ($n = 1000$)		*Step 1* only	*Step 1* and *Step 2*		Max number of states in DP_2	
Profit class	Weight type	#Opt (%)	Time (%)	Time (%)	Average (x10^3)	Max (x10^3)
$p1$	$a1$	72.5	54.0	46.0	0.1	0.2
	$a2$	60.0	56.8	43.2	0.1	0.5
$p2$	$a1$	43.3	50.6	49.4	0.8	16.0
	$a2$	49.2	49.4	50.6	1.6	20.5
$p3$	$a1$	69.2	58.1	41.9	2.8	56.2
	$a2$	52.5	78.0	22.0	1.8	6.8
$p4$	$a1$	48.3	78.0	22.0	2.1	19.6
	$a2$	57.5	73.7	26.3	2.6	14.9
$p5$	$a1$	22.5	43.0	57.0	5.1	36.1
	$a2$	24.2	43.5	56.5	13.4	58.9
$p6$	$a1$	82.5	41.5	58.5	4.9	25.2
	$a2$	27.5	36.0	64.0	7.6	47.8
$p7$	$a1$	59.2	51.0	49.0	0.9	3.2
	$a2$	58.3	48.5	51.5	2.3	7.5

Table 4.8 Numerical insights of the exact approach for instances with 1000 items.

Exact approach ($n = 10000$)		*Step 1* only	*Step 1* and *Step 2*		Max number of states in DP_2	
Profit class	Weight type	#Opt (%)	Time (%)	Time (%)	Average (x10^3)	Max (x10^3)
$p1$	$a1$	85.0	79.1	20.9	0.7	6.3
	$a2$	69.2	62.3	37.7	1.5	5.5
$p2$	$a1$	50.0	68.3	31.7	3.4	66.2
	$a2$	52.5	48.6	51.4	19.5	166.1
$p3$	$a1$	77.5	58.0	42.0	146.3	1115.8
	$a2$	56.7	90.1	9.9	45.8	247.4
$p4$	$a1$	73.3	74.5	25.5	119.9	713.0
	$a2$	79.2	83.8	16.2	139.8	885.8
$p5$	$a1$	27.5	39.3	60.7	164.5	1244.3
	$a2$	25.0	39.2	60.8	444.4	3088.3
$p6$	$a1$	83.3	32.0	68.0	187.5	700.5
	$a2$	33.3	26.8	73.2	321.6	1292.2
$p7$	$a1$	55.0	60.3	39.7	81.1	493.3
	$a2$	63.3	60.2	39.8	145.6	764.9

Table 4.9 Numerical insights of the exact approach for instances with 10000 items.

The results in the tables illustrate the effectiveness of the first step in solving PKP instances. Usually more than 50% of the instances are solved to optimality within this step. When both steps are involved, the computational

effort is on average equally distributed. We note however an increase of the percentages of the second step in classes $p5$ and $p6$. The number of states is in general reasonably limited allowing our algorithm to effectively solve all instances considered. The largest values of the number of states (with a maximum of about 3 millions) are reached in the instances with 10000 items.

The 0–1 Incremental Knapsack Problem

5.1 Introduction

We consider the 0–1 Incremental Knapsack Problem (IKP) as introduced in [37]. IKP is a generalization of the standard 0–1 Knapsack Problem (KP) where the capacity grows over T time periods. If an item is placed in the knapsack in a certain period, it cannot be removed afterwards. The problem calls for maximizing the sum of the profits over the whole time horizon.

IKP has many real-life applications since, from a practical perspective, it is often required in allocation resource problems to deal with changes in the input conditions and/or in a multi–period optimization framework. In [37], incremental versions of maximum flow, bipartite matching, and knapsack problems are introduced. The authors in [37] discuss the complexity of these problems and show how the incremental version even of a polynomial time solvable problem, like the max flow problem, turns out to be NP–hard. General techniques to adapt the algorithms for the considered optimization problems to their respective incremental versions are discussed. Also, a general purpose approximation algorithm is introduced. In [8], a PTAS is derived for IKP under the assumption $T = O(\sqrt{\log n})$, where n is the number of items. In addition, a constant factor algorithm is provided under mild restrictions on the growth rate of the knapsack capacity. The algorithm works also when discount factors are applied in each period. For further details on the matter, see [8].

In this chapter, we prove, first, the tightness of some approximation ratios derived in [37]. Then, we devise a PTAS for IKP when the number of time periods T can be considered as a constant. While this is a stronger assumption

than the one made for the PTAS in [8], our algorithm is much simpler and can be stated with less involved notation. Eventually, we consider the case where each item can be packed in the first period. Under this reasonable assumption, we manage to derive an algorithm with a constant approximation factor of $\frac{6}{7}$ when $T = 2$.

A journal version of the obtained results is currently being finalized. The remainder of the chapter is organized as follows. The linear programming formulation of the problem and the structure of its linear relaxation are described in Section 5.2. In Section 5.3, our approximation results are discussed.

5.2 Notation and problem formulation

In IKP a set of n items is given together with a knapsack with increasing capacity values c_t over time periods $t = 1, \ldots, T$. Each item i has a non–negative integer profit p_i and a non–negative integer weight w_i. The problem calls for maximizing the total profit of the selected items without exceeding the knapsack capacity over the given time horizon. If an item is placed in the knapsack, it cannot be removed at a later time. To derive an ILP-formulation, we associate with each item i a binary variable x_{it} such that $x_{it} = 1$ iff item i is placed in the knapsack in period t. IKP can be formulated according to the following ILP model (denoted by (IKP)):

(IKP):

$$\text{maximize} \quad \sum_{t=1}^{T} \sum_{i=1}^{n} p_i x_{it} \tag{5.1}$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_{it} \leq c_t \quad t = 1, \ldots, T; \tag{5.2}$$

$$x_{i(t-1)} \leq x_{it} \qquad i = 1, \ldots, n, \quad t = 2, \ldots, T; \tag{5.3}$$

$$x_{it} \in \{0, 1\} \qquad i = 1, \ldots, n, \quad t = 1, \ldots, T. \tag{5.4}$$

The objective function (5.1) maximizes the sum of the profits over the whole time horizon; the capacity constraints (5.2) guarantee that the items weights sum does not exceed the capacity c_t in each time period t; precedence constraints

(5.3) ensure that if an item is chosen at time t, it will be not removed in successive periods; finally, constraints (5.4) indicate that all variables are binary. We will denote the optimal solution value of model (IKP) by z^*.

For each period t and the related capacity value c_t, we define the corresponding standard knapsack problem as KP_t. This means that in KP_t we consider only one of the T constraints (5.2). Finally, we remark that the linear relaxation of model (IKP), where constraints (5.4) are replaced by the inclusion in the interval $[0,1]$, can be easily computed. In fact, it suffices to order the items by non–increasing $\frac{p_i}{w_i}$ in $O(n\log n)$ and to fill the capacity of the knapsack in each period according to this ordering in $O(n)$. The execution time for solving the linear relaxation of IKP, hereafter denoted by IKP^{LP}, is thus $O(n\log n)$.

5.3 Approximating IKP

5.3.1 Approximation ratios of a general purpose algorithm

In [37], a general framework for deriving approximation algorithms is provided. Following the scheme in [37] for IKP, we consider the following algorithm A. The algorithm employs an ε–approximation scheme to obtain a feasible solution for each knapsack problem KP_t. Denote the corresponding solution value by z_t^A for $t=1,\ldots,T$. Each such solution is also a feasible solution for IKP where z_t^A is present in all successive time periods. The algorithm chooses as a solution value z^A the maximum among all these candidates, i.e.

$$z^A = \max_{t=1,\ldots,T}\{(T-t+1)z_t^A\}. \tag{5.5}$$

The following Theorem which is a reformulation of Theorem 3 in [37] holds.

Theorem 10. *Algorithm A is an approximation algorithm for IKP with ratio bounded by $\frac{1-\varepsilon}{\mathcal{H}_T}$, where $\mathcal{H}_T$ is the harmonic number $1+\frac{1}{2}+\cdots+\frac{1}{T}$.*

The authors in [37] proved the tightness of this approximation bound for the incremental max flow problem. For IKP, the tightness of the bound is an open question. In this contribution, we consider the case where algorithm A solves each KP_t to optimality (using e.g. dynamic programming by weights or the *Combo* algorithm) and thus provides an approximation ratio of $\frac{1}{\mathcal{H}_T}$. We manage to prove the tightness of this approximation bound by an alternative analysis of the performance of the algorithm based on a Linear Programming (LP) model. More precisely, we consider an LP formulation with non-negative variables h^A and h_t associated with z^A and z_t^A respectively and a positive parameter $OPT > 0$ associated with z^*. The corresponding LP model for evaluating the worst case performance of algorithm A is as follows:

$$\text{minimize} \quad h^A \tag{5.6}$$

$$\text{subject to} \quad h^A \geq (T-t+1)h_t \quad t=1,\ldots,T; \tag{5.7}$$

$$\sum_{t=1}^{T} h_t \geq OPT \tag{5.8}$$

$$h^A \geq 0 \tag{5.9}$$

$$h_t \geq 0 \quad t=1,\ldots,T. \tag{5.10}$$

The value of the objective function (5.6) provides a lower bound on the worst case performance of algorithm A. Constraints (5.7) guarantee that the contribution of each knapsack problem KP_t as a solution of IKP will be taken into account according to (5.5). Constraint (5.8) indicates that the sum of the optimal KPs solution values z_t^A over all T knapsack problems constitute a trivial upper bound on z^*. Constraints (5.9, 5.10) indicate that variables are non-negative. We will denote the optimal value of h^A and h_t $(t=1,\ldots,T)$ by h^{A^*} and h_t^* respectively. By setting parameter OPT to an arbitrary positive value, the corresponding lower bounds on the performance ratio of algorithm A for any T are given by $\frac{h^{A^*}}{OPT}$.

Model (5.6)–(5.10) allows us to prove the tightness of the approximation bound $\frac{1}{\mathcal{H}_T}$ of algorithm A.

Theorem 11. *For any value of T, if algorithm A solves to optimality each KP_t $(t = 1,\ldots,T)$, the approximation ratio $\frac{1}{\mathcal{H}_T}$ of the algorithm is tight for IKP.*

Proof. We first provide a characterization of the optimal solution of model (5.6)–(5.10) and show that, for any T, the bound $\frac{h^{A^*}}{OPT}$ is actually equal to $\frac{1}{\mathcal{H}_T}$. Given the constraints in the model, we note that the optimal value h^{A^*} will be naturally equal to at least one of the right–hand side values $(T-t+1)h_t^*$ of constraints (5.7). Also, the optimal solution will always fulfill $\sum_{t=1}^{T} h_t^* = OPT$. Suppose by contradiction that there is an optimal solution with $\sum_{t=1}^{T} h_t^* > OPT$. In such a case, we could always decrease h^{A^*} by jointly decreasing the corresponding h_t^* values (i.e. such that $h^{A^*} = (T-t+1)h_t^*$), thus contradicting the optimality of the solution.
Moreover, in the optimal solution all right–hand side values of constraints (5.7) will reach the same value. Suppose again by contradiction that there exists an optimal solution where this structure does not hold, i.e. there are two time periods t' and t'' with $(T-t'+1)h_{t'} = \max_t\{(T-t+1)h_t\} > (T-t''+1)h_{t''}$. In this case, we could lower the objective by decreasing $h_{t'}$ and increasing $h_{t''}$ by the same value thus preserving the equality in (5.8) but allowing a decrease of h^A, which contradicts the claim.
Based on this structural property, computing the optimal solution of the LP model amounts to solving the following system with $T+1$ equations in $T+1$ unknowns inducing a unique solution:

$$\begin{cases} h_t^* = \frac{h^{A^*}}{T-t+1} \quad t = 1,\ldots,T \\ \sum_{t=1}^{T} h_t^* = OPT \end{cases} \tag{5.11}$$

Indeed, by combining the first T equations with the latter one, correspondingly, we have that

$$\frac{h^{A^*}}{T} + \frac{h^{A^*}}{T-1} + \cdots + h^{A^*} = \mathcal{H}_T h^{A^*} = OPT \implies h^{A^*} = \frac{OPT}{\mathcal{H}_T} \tag{5.12}$$

that is $\frac{h^{A^*}}{OPT} = \frac{1}{\mathcal{H}_T}$. To prove the tightness of the bound $\frac{1}{\mathcal{H}_T}$, notice from (5.11) and (5.12) that we have

$$h_t^* = \frac{OPT}{\mathcal{H}_T(T-t+1)} \quad t = 1,\dots,T. \tag{5.13}$$

Then, it suffices to derive instances where the optimal solution values of KPs in each period are equal to h_t^* $(t = 1,\dots,T)$ and the optimal solution value for IKP is equal to the sum of all these solutions, namely $z^* = \sum_{t=1}^{T} h_t^*$. Such target instances can be generated by the following procedure:

1. We first represent the harmonic number $\mathcal{H}_T$ as a fraction, i.e. $\mathcal{H}_T = \frac{a}{b}$ where b is the smallest common multiple of the denominators of the fractions $\frac{1}{2} + \cdots + \frac{1}{T}$. Then, we set $OPT = a$ and solve model (5.6)–(5.10) according to (5.12)–(5.13). This setting guarantees to get integer h_t^*.

2. After that, we generate an IKP instance with: $n = b$, $p_j = w_j = 1$ $(j = 1,\dots,n)$, $c_t = h_t^*$ $(t = 1,\dots,T)$. The optimal solution of each KP_t will pack items until the corresponding capacity c_t is fulfilled and thus will yield a solution value equal to h_t^*. The number of items is b because the capacity in the last period T is $c_T = h_T^* = \frac{OPT}{\mathcal{H}_T(T-T+1)} = \frac{a}{\frac{a}{b}} = b$. At the same time, the optimal solution for IKP can be obtained by progressively packing all items over time periods while fulfilling the capacities c_t, hence $z^* = \sum_{t=1}^{T} c_t = \sum_{t=1}^{T} h_t^*$.

□

As an example of the outlined procedure, consider the case with $T = 3$ for which $\mathcal{H}_T = \frac{11}{6}$. We solve model (5.6)–(5.10) by setting $OPT = 11$. Then, the following IKP instance is generated:

$$n = 6,\ p_j = w_j = 1\ (j = 1,\dots,6),\ c_1 = 2,\ c_2 = 3,\ c_3 = 6.$$

The optimal solution of IKP is given by packing all items over time periods and fulfilling the corresponding capacities ($z^* = 11$). The optimal solutions values of the KPs are equal to 2, 3, 6 respectively. Hence we have

$z^A = \max\{3*2, 2*3, 6\} = 6$ which proves the tightness of the approximation bound $\frac{1}{\mathcal{H}_3} = \frac{6}{11}$.

We remark that the bound tightness cannot be straightforwardly generalized when a ε–approximation scheme is adopted for solving each KP. We could get the ratio in Theorem 10 by solving model (5.6)–(5.10) where the term $\sum_{t=1}^{T} h_t$ in constraint (5.8) is divided by $(1-\varepsilon)$. However, the generation of tight instances is strictly related to the choice of the approximation algorithm for KPs.

5.3.2 A PTAS when T is a constant

Similarly to the line of reasoning for deriving PTAS's for KP (see, e.g., [84, 12]), we propose an approximation scheme for IKP based on guessing the k items with largest profits in an optimal solution. We first define the following variant of algorithm A described in Section 5.3.1, denoted as algorithm A'. We run an FPTAS for each time period t yielding ε-approximations $z_t^{A'}$. Then we also consider an alternative solution for IKP derived by computing the optimal solution of IKP^{LP} and rounding down all fractional variables to 0. Thus, we get a feasible solution for IKP with solution value z'. Finally, we take the maximum between these $T+1$ candidates reaching a solution value $z^{A'}$, namely

$$z^{A'} = \max\{z', \max_t\{(T-t+1)z_t^{A'}\}\}. \tag{5.14}$$

Since computing z' requires $O(n \log n)$ and the running time of the FPTAS for KP can be bounded by $O(n\log(\frac{1}{\varepsilon}) + (\frac{1}{\varepsilon})^3 \log^2(\frac{1}{\varepsilon}))$ (see [45, 46]) the overall running time of algorithm A' is

$$O(n \log n + T(n \log(\frac{1}{\varepsilon}) + (\frac{1}{\varepsilon})^3 \log^2(\frac{1}{\varepsilon}))).$$

A useful property of algorithm A' is the following. Because of the special structure of IKP^{LP}, at most T fractional variables will be rounded down to

get z'. Thus, we have that

$$z^* \leq z' + Tp_{max} \leq z^{A'} + Tp_{max} \tag{5.15}$$

where p_{max} is the maximum profit of any item.
The overall approximation ratio of algorithm A', denoted by ρ, can be stated by considering the solution values $z_t^{A'}$ in each time period. Hence, according to Theorem 10, we have $\rho = \frac{1-\varepsilon}{\mathcal{H}_T}$.

We can now state our approximation scheme, denoted by algorithm *Approx*, as follows:

1. We first sort the items by decreasing efficiency $\frac{p_j}{w_j}$. We then guess the k items with largest profits in an optimal solution as well as how these k items are distributed over the T time periods. This corresponds to consider $O(n^k)$ choices for the items together with $O(k^T)$ possible choices for their distributions over time.

2. For each feasible distribution of the items, we then consider the remaining IKP instance. We indicate by $P(k)$ the overall profit contribution of the k items. We also denote by c_t^R the residual capacities after the insertions of the items in each time period t.

3. In order to maintain the incremental structure of the problem, we set $c_t^R = \min\{c_t^R, c_{t+1}^R\}$ in decreasing order of t, i.e. for $t = T-1, \ldots, 1$. The corresponding residual IKP instance is denoted by R.

4. Given the ordering of the items, we apply algorithm A' to instance R getting a solution value $z_R^{A'}$. The sum $z_R^{A'} + P(k)$ yields the corresponding solution value for IKP.

5. The overall best solution over all choices of k with value z^{Approx} is eventually returned.

When the input T is considered as a constant, the outlined algorithm constitutes a PTAS for IKP.

Theorem 12. *Algorithm Approx is a PTAS for IKP when T is a constant.*

Proof. We first show that the algorithm is an ε–approximation scheme. We consider two cases depending on a parameter $f \in (0,1)$ and analyze the iteration where the optimal distribution of the items with largest profits is selected. In the following, we will denote by z_R^* and by p_{max}^R the optimal solution value and the maximum profit of the items in instance R.

Case 1: $P(k) \geq f \cdot z^*$
When the optimal distribution of the k items with largest profits is considered, we have $z^* = z_R^* + P(k)$. Hence, the following series of inequalities holds:

$$\begin{aligned} z^{Approx} &\geq P(k) + z_R^{A'} \geq P(k) + \rho z_R^* \\ &= P(k) + \rho[z^* - P(k)] = (1-\rho)P(k) + \rho z^* \\ &\geq (1-\rho) f z^* + \rho z^* = [(1-\rho)f + \rho] z^* \end{aligned} \tag{5.16}$$

Case 2: $P(k) < f \cdot z^*$
Since p_{max}^R is less than (or equal to) the minimum profit of the k items which in turn is less than $\frac{1}{k} f \cdot z^*$ (consider that each of the k selected items contributes to $P(k)$ at least once with its profit), we have that

$$p_{max}^R < \frac{1}{k} f z^*. \tag{5.17}$$

Then, the following series of inequality holds:

$$z^* = P(k) + z_R^* \leq P(k) + z_R^{A'} + T p_{max}^R \leq z^{Approx} + T \frac{1}{k} f z^* \tag{5.18}$$

The first inequality comes from (5.15). The second inequality derives from the fact that $P(k) + z_R^{A'} \leq z^{Approx}$ and (5.17). We have from (5.18) that

$$z^{Approx} \geq (1 - T \frac{1}{k} f) z^*. \tag{5.19}$$

Now, given ε and ρ, we set k and f as follows:

$$k := \left\lceil \frac{T}{\varepsilon} \right\rceil \tag{5.20}$$

$$f := 1 - \frac{\varepsilon}{1-\rho} = \frac{1-\rho-\varepsilon}{1-\rho} \tag{5.21}$$

We easily note that $f < 1$ and we also have $f > 0$ since $\rho = \frac{1-\varepsilon}{\mathcal{H}_T}$ and thus $1-\varepsilon > \rho$.
For *Case 1*, plugging the value of f in (5.16) yields

$$\begin{aligned} z^{Approx} &\geq [(1-\rho)f + \rho]z^* \\ &= [(1-\rho-\varepsilon) + \rho]z^* = (1-\varepsilon)z^*. \end{aligned} \tag{5.22}$$

For *Case 2*, we plug (5.20) in (5.19) and get

$$z^{Approx} \geq (1-\varepsilon f)z^* \geq (1-\varepsilon)z^*. \tag{5.23}$$

Hence, algorithm *Approx* is an ε–approximation scheme. The running time is given by the initial sorting of the items and by the running time of algorithm A' (without the sorting contribution) multiplied by $O(n^k k^T)$. The overall time complexity of the approximation scheme is

$$O(n\log n + n^{\lceil \frac{T}{\varepsilon} \rceil}(\left\lceil \frac{T}{\varepsilon} \right\rceil)^T T(n\log(\frac{1}{\varepsilon}) + (\frac{1}{\varepsilon})^3 \log^2(\frac{1}{\varepsilon}))) \tag{5.24}$$

which establishes a PTAS for IKP when T is a constant. □

We remark that [8] introduces a PTAS for $T = O(\sqrt{\log n})$ thus providing a stronger theoretical result in terms of approximation schemes. Nonetheless, the PTAS in [8] relies on solving a large number, namely $O(n(\frac{1}{\varepsilon}+T)^{O(\log(\frac{T}{\varepsilon})/\varepsilon^2)})$, of non–trivial LP models. Our approach instead does not require the solution of any complicated LP model and it can be also stated with less involved notation. Hence, from a practical perspective, our PTAS may constitute an appealing approximation algorithm for reasonable values of T. A representative comparison of the performance of the algorithms is provided in Figure 5.1. The Figure plots the complexity of *Approx* algorithm and the number of the LP models required in the PTAS in [8] for different values of n $(= 100, 1000, 10000)$

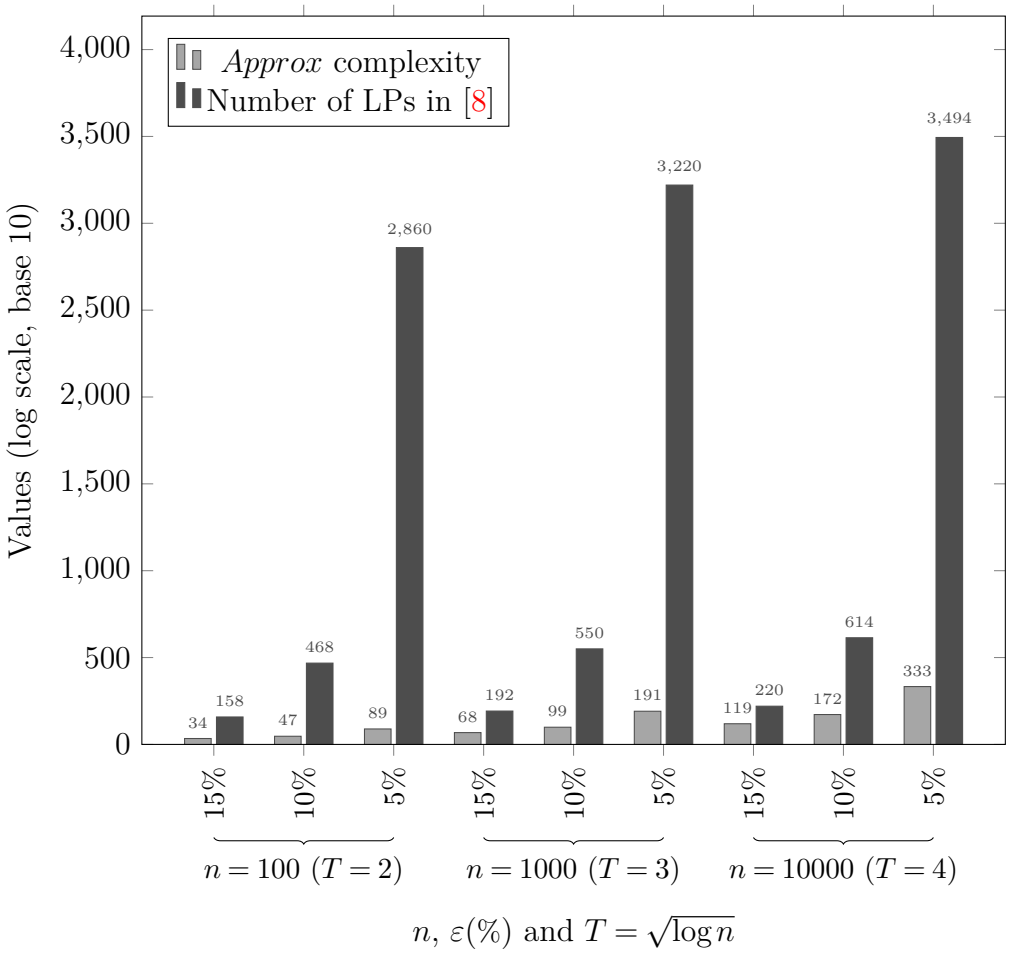

Figure 5.1 Numerical comparison between the complexity of *Approx* algorithm and the number of LPs required by the PTAS in [8]. Tests with $n = 100, 1000, 10000$ (resp. $T = 2, 3, 4$), $\varepsilon = 0.15, 0.1, 0.05$.

and ε $(= 0.15, 0.1, 0.05)$. The number of period is $T = \sqrt{\log n}$ according to the assumption in [8]. As shown in Figure 5.1, the differences are remarkable even without considering the execution time for solving each LP.

5.3.3 A constant factor algorithm for a restricted variant

We provide a constant factor approximation algorithm for IKP with two periods $(T = 2)$ under the mild assumption that each item can be packed in the first period, i.e. $w_i \leq c_1$ $(i = 1, \ldots, n)$. The algorithm, denoted as H, is based on the

optimal solutions of the two knapsack problems KP_1 and KP_2 associated with each of the two time periods. The algorithm can be outlined as follows:

1. We solve to optimality KP_1. We then solve KP_2 where the optimal solution set of items in KP_1 is placed inside the knapsack.
2. As an alternative solution, we first solve to optimality KP_2. Then, we consider the optimal set of items in KP_2 and solve KP_1 with these items only.
3. The best solution found is returned.

We again rely on Linear Programming to derive the approximation ratio of algorithm H. First, let us define the following subsets of items S_i:

- S_1: subset of items included both in the optimal solutions of KP_1 and KP_2;
- S_2: remaining subset of items in the optimal solution of KP_1;
- S_3: remaining subset of items not exceeding capacity c_1 in the optimal solution of KP_2;
- S_4: first item exceeding c_1 in the optimal solution of KP_2;
- S_5: remaining subset of items in the optimal solution of KP_2.

We will denote the sum of the profits and weights of the items included in subset S_i by P_i and W_i respectively.
The union of S_1 and S_2 constitutes the optimal solution set of items of KP_1. Likewise, the union of S_1, S_3, S_4 and S_5 represents the optimal solution set of KP_2. Figure 5.2 illustrates the decomposition of the optimal solution sets in each time period. The dashed lines in Figure 5.2 refer to the item in S_4 which exceeds the first capacity value.
According to the above definitions, we have the following inequalities

$$W_1 + W_2 \leq c_1; \tag{5.25}$$

$$W_1 + W_3 \leq c_1; \tag{5.26}$$

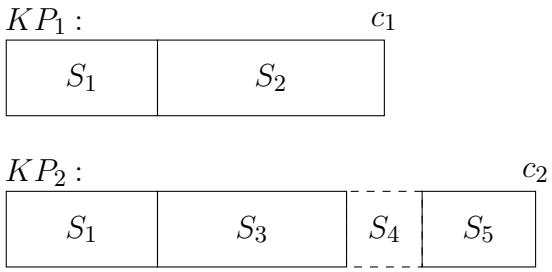

Figure 5.2 Decomposition of the optimal solutions of KP_1 and KP_2.

$$W_1 + W_3 + W_4 > c_1; \tag{5.27}$$

$$W_1 + W_3 + W_4 + W_5 \leq c_2; \tag{5.28}$$

$$W_1 + W_2 + W_5 < c_2. \tag{5.29}$$

Inequality (5.29) derives directly from inequalities (5.25), (5.27) and (5.28). The optimal solution values of KP_1 and KP_2 are equal to $P_1 + P_2$ and $P_1 + P_3 + P_4 + P_5$ respectively. Now we can state three feasible solutions reached by algorithm H:

- The optimal solution set of KP_1 in the two periods plus the additional packing of items in S_5 in the second period. The whole profit is: $2(P_1 + P_2) + P_5$;
- the optimal solution set of KP_2 in the second period with the packing of items in subsets S_1 and S_3 in the first period. The resulting profit is: $2(P_1 + P_3) + P_4 + P_5$;
- the optimal solution set of KP_2 in the second period with item S_4 placed in the knapsack in the first period. The profit of this solution is: $P_1 + P_3 + 2P_4 + P_5$.

Algorithm H will return a solution at least as good as the best of the above three solutions. In order to evaluate the worst case performance of the heuristic, we consider an LP formulation where we associate a non–negative variable h with the solution value computed by the algorithm. In addition, the profits of the subsets S_i are associated with non–negative variables $\bar{p}_i$ $(i = 1, \ldots, 5)$. As in Section 5.3.1, the positive parameter OPT represents z^*. This implies the

following LP model:

$$\begin{aligned}
\text{minimize} \quad & h && (5.30)\\
\text{subject to} \quad & h \geq 2(\bar{p}_1 + \bar{p}_2) + \bar{p}_5 && (5.31)\\
& h \geq 2(\bar{p}_1 + \bar{p}_3) + \bar{p}_4 + \bar{p}_5 && (5.32)\\
& h \geq \bar{p}_1 + \bar{p}_3 + 2\bar{p}_4 + \bar{p}_5 && (5.33)\\
& (\bar{p}_1 + \bar{p}_2) + (\bar{p}_1 + \bar{p}_3 + \bar{p}_4 + \bar{p}_5) \geq OPT && (5.34)\\
& h \geq 0 && (5.35)\\
& \bar{p}_i \geq 0 \quad i = 1, \ldots, 5. && (5.36)
\end{aligned}$$

The value of the objective function (5.30) represents a lower bound on the worst case performance of algorithm H. Constraints (5.31)–(5.33) guarantee that the algorithm will select the best of the three feasible solutions constructed as described above. Constraint (5.34) is due to the fact that the sum of the optimal values of KP_1 and KP_2 is an upper bound on z^*. Constraints (5.35), (5.36) indicate that the variables are non-negative.

Setting the parameter $OPT = 1$, we get an optimal value h^* equal to $0.8571\ldots = \frac{6}{7}$. Thus, a lower bound on the performance ratio provided by algorithm H is equal to $\frac{h^*}{OPT} = \frac{6}{7}$.

We can show that the ratio of $\frac{6}{7}$ is tight by considering an instance with $n = 5$, $c_1 = 3 + \delta$, $c_2 = 4$ (with $\delta > 0$ being an arbitrary small number) and the following entries:

i	1	2	3	4	5
p_i	3	2	2	$1-\delta$	$1-\delta$
w_i	$3+\delta$	2	2	1	1

The optimal solution of KP_1 consists of item 1. The corresponding IKP solution considers only item 1 in the second period with a total profit of 6. The optimal solution of KP_2 consists in packing items 2 and 3. In the corresponding IKP solution either item 2 or item 3 is placed in the knapsack in the first period. The resulting profit is again 6. An optimal solution of IKP selects items 2 and 4 in the first period together with item 5 in the second period, thus $z^* = 7 - 3\delta$.

The approximation ratio of algorithm H is

$$\frac{\max\{6,6\}}{7-3\delta} = \frac{6}{7-3\delta}$$

which can be arbitrarily close to $\frac{6}{7}$ as the value of δ goes to 0. Hence, we can state the following theorem.

Theorem 13. *Algorithm H is an approximation algorithm for IKP with a tight approximation ratio of $\frac{6}{7}$.*

Algorithm H is not a polynomial time algorithm since requires the optimal solutions of KP_1 and KP_2. We could solve these knapsack problems by an ε–approximation scheme (PTAS or FPTAS) to get a polynomial running time at the cost of a decrease of the approximation bound. The following Corollary shows that the bound is decreased by a factor $(1-\varepsilon)$.

Corollary 2. *If an ε–approximation scheme is employed for solving the standard knapsack problems KP_1 and KP_2, the approximation ratio of algorithm H is bounded by $\frac{6}{7}(1-\varepsilon)$.*

Proof. We consider again model (5.30)–(5.36) to evaluate the performance of the algorithm. If the subsets S_i $(i = 1,\ldots,5)$ correspond to the items of the ε–approximations for KP_1 and KP_2, constraints (5.31)–(5.33) straightforwardly hold. Then, we just replace constraint (5.34) by constraint

$$(\bar{p}_1+\bar{p}_2)+(\bar{p}_1+\bar{p}_3+\bar{p}_4+\bar{p}_5) \geq OPT(1-\varepsilon) \tag{5.37}$$

which indicates that the sum of the approximate solutions divided by $(1-\varepsilon)$ provides an upper bound on the optimal solution value of IKP. Since we get $h^* = 6/7$ when we set $OPT(1-\varepsilon) = 1$, a lower bound on the approximation ratio of algorithm H is equal to $\frac{h^*}{OPT} = \frac{\frac{6}{7}}{\frac{1}{1-\varepsilon}} = \frac{6}{7}(1-\varepsilon)$.

□

Conclusions and Future Developments

In this thesis we dealt with four generalizations of the classical Knapsack Problem (KP) involving side constraints beyond the capacity bound. We first presented an overview of the research pursued in the literature for KP in Chapter 1. Then, we considered the 0–1 Knapsack Problem with Setups (KPS), the 0–1 Collapsing Knapsack Problem (CKP), the 0–1 Penalized Knapsack Problem (PKP) and the 0–1 Incremental Knapsack Problem (IKP).

In Chapter 2, we proposed an exact enumerative approach for KPS based on an effective exploration of a specific set of variables that leads to solve standard knapsack problems. The presented approach proves to be very effective and capable of handling instances with up to 100000 items and 200 families with little computational effort while previous approaches were limited to instances with up to 10000 items. The approach outperforms CPLEX 12.5 and favorably compares to the algorithms available in literature. We also devised an improved dynamic programming algorithm by effectively leveraging a method from the literature applied for a related knapsack problem.
Then, we derived a general non-approximability result. To gain further insights into the structural difficulty of KPS, we investigated several relevant special cases of KPS arising from certain additional, but plausible restrictions on the input data. We managed to derive several approximation algorithms and resulting fully polynomial time approximation schemes (FPTASs). In this way we narrowed the gap between approximable (in the sense of existence of an FPTAS) and inapproximable cases.

In Chapter 3, we presented a novel 0–1 linear programming formulation for CKP and an efficient method for tackling CKP and the related multi-dimensional variant. The novel formulation of CKP shows to be very effective when solved by CPLEX 12.5 and manages to handle instances of much larger size than the ones considered by the approaches available in the literature. A reduction procedure based on solving continuous problems drastically limits the solution space and significantly enhances the performance of the ILP solver. An efficient exact approach applying an original branching scheme to the proposed ILP formulation is able to further improve the results. Our approach is capable of finding optimal solutions for instances with up to 100000 items, while previous approaches were limited to 1000 items. We considered also M-CKP involving M capacity constraints. The proposed exact approach turns out to be very effective for 2-CKP, where the approach showed up to solve to optimality with limited time instances with up 100000 items. In other multidimensional variants with up to 5 capacity constraints, the exact method globally outperforms CPLEX 12.5 standalone and CPLEX 12.5 launched in cascade after the application of the reduction procedure, even though the size of the instances solved to optimality significantly decreases when M increases.

In Chapter 4, we proposed a dynamic programming based exact approach for PKP which leverages an algorithmic framework originally constructed for KP. The proposed approach turns out to be very effective in solving instances of the problem with up to 10000 items and it favorably compares to both solver CPLEX 12.5 and an exact algorithm in the literature. We also gave further insights on the structure and properties of PKP by providing a characterization of its linear relaxation, an effective procedure to compute upper bounds on the problem and a negative approximation result.

In Chapter 5, we devised a series of results for IKP extending the contributions in the literature. At first, we managed to prove approximation ratios of a general purpose algorithm previously laid out. Secondly, we derived approximation algorithms under reasonable restrictions on the input data. We established a polynomial time approximation scheme (PTAS) when one of the problem inputs can be considered as a constant and we introduced an algorithm with an approximation ratio of $\frac{6}{7}$ for a variant involving two time periods.

The research contributions provided in the thesis can be extended in different directions.

In KPS and CKP we exploited the partitioning of the variables set into two levels. Our general algorithmic idea relies on inducing problems tractable in practice through an effective exploration of the solution space of first level variables. A possible future development would be to adapt our methods to other knapsack–like problems or to optimization problems involving two sets of variables. Besides, it would be interesting to study further adaptations of algorithmic frameworks for KP, as we did for PKP, to other relevant KP variants.

We derived negative approximation results by exploiting the presence of variables with positive and negative impacts in the objective functions of KPS and PKP. When this general condition holds, we may investigate whether our procedures for establishing inapproximability results could be extended to other optimization problems.

To the author's knowledge, no computational experience has been provided for IKP so far. Thus, in future contributions we could design new solution approaches for the problem and test their performance after generating benchmark and challenging to solve instances.

Another appealing research topic could be the tolerance (or stability) analysis. This analysis studies the robustness of the optimal solutions of integer programming problems when perturbations of the input data occur. A paper recently appeared ([80]) introduces effective techniques to perform a tolerance analysis for KP. In this respect, we could investigate possible extensions of the results provided in [80] to the considered knapsack problems.

After completion of the thesis, we became aware of two recent published papers at SODA 2017 conference ([11], [48]). The papers introduce new approaches and improved pseudopolynomial algorithms for the Subset Sum Problem. It might be also interesting to evaluate applications of the techniques employed in [11, 48] to the problems tackled in the thesis.

Bibliography

[1] Flexmeter. http://flexmeter.polito.it.

[2] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job–shop scheduling. *Management Science*, 34:391–401, 1988.

[3] U. Akinc. Approximate and exact algorithm for the fixed–charge knapsack problem. *European Journal of Operational Research*, 170:363–375, 2004.

[4] N. Altay, P. E. Robinson Jr, and K. M. Bretthauer. Exact and heuristic solution approaches for the mixed integer setup knapsack problem. *European Journal of Operational Research*, 190:598–609, 2008.

[5] E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.

[6] E. Balas and E. Zemel. An algorithm for large zero–one knapsack problems. *Operations Research*, 28:1130–1154, 1980.

[7] R. E. Bellman. *Dynamic programming*. Princeton University Press, 1957.

[8] D. Bienstock, J. Sethuraman, and C. Ye. Approximation algorithms for the incremental knapsack problem via disjunctive programming, 2013. Available in: `arXiv:1311.4563`.

[9] S. Boussier, M. Vasquez, Y. Vimont, S. Hanafi, and P. Michelon. A multi-level search strategy for the 0-1 multidimensional knapsack problem. *Discrete Applied Mathematics*, 158:97–109, 2010.

[10] E. F. Brickell and A. M. Odlyzko. Cryptanalysis: A survey of recent results. In G.J. Simmons, editor, *Contemporary Cryptology*, pages 501–540. IEEE Press, 1991.

[11] K. Bringmann. A near–linear pseudopolynomial time algorithm for subset sum. In *Proceedings of the Twenty–Eighth Annual ACM–SIAM Symposium on Discrete Algorithms*. 2017.

[12] A. Caprara, H. Kellerer, U. Pferschy, and D. Pisinger. Approximation algorithms for knapsack problems with cardinality constraints. *European Journal of Operational Research*, 123:333–345, 2000.

[13] J. Carlier. The one–machine sequencing problem. *European Journal of Operational Research*, 11:42–47, 1982.

[14] M. Caserta, E. Quinonez Rico, and A. Marquez Uribe. A cross entropy algorithm for the knapsack problem with setups. *Computers & Operations Research*, 35:241–252, 2008.

[15] A. Ceselli and G. Righini. An optimization algorithm for a penalized knapsack problem. *Operations Research Letters*, 34:394–404, 2006.

[16] E. D. Chajakis and M. Guignard. Exact algorithms for the setup knapsack problem. *INFOR*, 32:124–142, 1994.

[17] K. Chebil and M. Khemakhem. A dynamic programming algorithm for the knapsack problem with setup. *Computers & Operations Research*, 64:40–50, 2015.

[18] G. Cho and D. X. Shaw. A depth–first dynamic programming algorithm for the tree knapsack problem. *INFORMS Journal on Computing*, 9:431–438, 1997.

[19] P. C. Chu and J. E. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4:63–86, 1998.

[20] G. Costanza. Algoritmi per il waste management nel paradigma internet of things, 2016. Master of science thesis in Industrial Engineering and Management, Politecnico di Torino.

[21] G. B. Dantzig. Discrete–variable extremum problems. *Operations Research*, 5:266–277, 1957.

[22] F. Della Croce and A. Grosso. Simplex algorithms for linear programming. In *Concepts of Combinatorial Optimization*, pages 135–155. Wiley, 2010.

[23] F. Della Croce and A. Grosso. Improved core problem based heuristics for the 0/1 multi-dimensional knapsack problem. *Computers & Operations Research*, 39:27–31, 2012.

[24] F. Della Croce and F. Salassa. Improved lp-based algorithms for the closest string problem. *Computers & Operations Research*, 39:746–749, 2012.

[25] F. Della Croce, F. Salassa, and R. Scatamacchia. An exact approach for the 0–1 knapsack problem with setups. *Computers & Operations Research*, 80:61–67, 2017.

[26] F. Della Croce, F. Salassa, and R. Scatamacchia. A new exact approach for the 0–1 collapsing knapsack problem. *European Journal of Operational Research*, 260:56–69, 2017.

[27] R. S. Dembo and P. L. Hammer. A reduction algorithm for knapsack problems. *Methods of Operations Research*, 36:49–60, 1980.

[28] K. Dudziński and S. Walukiewicz. Exact methods for the knapsack problem and its generalizations. *European Journal of Operational Research*, 28:3–21, 1987.

[29] D. Fayard and G. Plateau. An algorithm for the solution of the 0–1 knapsack problem. *Computing*, 28:269–287, 1982.

[30] D. Fayard and G. Plateau. An exact algorithm for the 0–1 collapsing knapsack problem. *Discrete Applied Mathematics*, 49:175–187, 1994.

[31] C. E. Ferreira, A. Martin, C. C. de Souza, R. Weismantel, and L. A. Wolsey. Formulations and valid inequalities for the node capacitated graph partitioning problem. *Mathematical Programming*, 74:247–266, 1996.

[32] F. V. Fomin and D. Kratsch. *Exact Exponential Algorithms*. Springer, 2010.

[33] A. Fréville. The multidimensional 0–1 knapsack problem: an overview. *European Journal of Operational Research*, 155:1–21, 2004.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman, 1979.

[35] F. Glover. A multiphase dual algorithm for the zero–one integer programming problem. *Operations Research*, 13:879–919, 1965.

[36] P. L. Hammer, E. L. Johnson, and U. N. Peled. Facet of regular 0–1 polytopes. *Mathematical Programming*, 8:179–206, 1975.

[37] J. Hartline and A. Sharp. An incremental model for combinatorial maximization problems. In Carme Àlvarez and María J. Serna, editors, *WEA, volume 4007 of Lecture Notes in Computer Science*, pages 36–48. Springer, 2006.

[38] C. A. R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[39] K. L. Hoffman and M. Padberg. Solving airline crew scheduling problems by branch–and–cut. *Management Science*, 39:657–682, 1993.

[40] E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21:277–292, 1974.

[41] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22:463–468, 1975.

[42] H. Iida and T. Uno. A short note on the reducibility of the collapsing knapsack problem. *Journal of the Operations Research Society of Japan*, 45:293–298, 2002.

[43] G. P. Ingargiola and J. F. Korsh. Reduction algorithm for zero–one single knapsack problems. *Management Science*, 20:460–463, 1973.

[44] D. S. Johnson and K. A. Niemi. On knapsacks, partitions, and a new dynamic programming technique for trees. *Mathematics of Operations Research*, 8:1–14, 1983.

[45] H. Kellerer and U. Pferschy. A new fully polynomial time approximation scheme for the knapsack problem. *Journal of Combinatorial Optimization*, 3:59–71, 1999.

[46] H. Kellerer and U. Pferschy. Improved dynamic programming in connection with an FPTAS for the knapsack problem. *Journal of Combinatorial Optimization*, 8:5–11, 2004.

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

[48] K. Koiliaris and C. Xu. A faster pseudopolynomial time algorithm for subset sum. In *Proceedings of the Twenty–Eighth Annual ACM–SIAM Symposium on Discrete Algorithms*. 2017.

[49] P. J. Kolesar. A branch and bound algorithm for the knapsack problem. *Management Science*, 13:723–735, 1967.

[50] G. Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59:345–358, 1992.

[51] E. L. Lawler. Fast approximation algorithms for knapsack problems. *Mathematics of Operations Research*, 4:339–356, 1979.

[52] Ch. Lenté, M. Liedloff, A. Soukhal, and V. T'Kindt. On an extension of the sort & search method with application to scheduling theory. *Theoretical Computer Science*, 511:13–22, 2013.

[53] E. Lin. A bibliographical survey on some well–known non–standard knapsack problems. *INFOR*, 36:274–317, 1998.

[54] A. Lodi, S. Martello, and M. Monaci. Two–dimensional packing problems: a survey. *European Journal of Operational Research*, 141:241–252, 2002.

[55] M. J. Magazine and O. Oguz. A fully polynomial approximation algorithm for the 0–1 knapsack problem. *European Journal of Operational Research*, 8:270–273, 1981.

[56] R. Mansini and M. G. Speranza. Coral: An exact algorithm for the multidimensional knapsack problem. *INFORMS Journal on Computing*, 24:399–415, 2012.

[57] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0–1 knapsack problem. *Management Science*, 45:414–424, 1999.

[58] S. Martello, D. Pisinger, and P. Toth. New trends in exact algorithms for the 0–1 knapsack problems. *European Journal of Operational Research*, 123:325–332, 2000.

[59] S. Martello and P. Toth. An upper bound for the zero–one knapsack problem and a branch and bound algorithm. *European Journal of Operational Research*, 1:169–175, 1977.

[60] S. Martello and P. Toth. A new algorithm for the 0–1 knapsack problem. *Management Science*, 34:633–644, 1988.

[61] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, 1990.

[62] S. Martello and P. Toth. Upper bounds and algorithms for hard 0–1 knapsack problems. *Operations Research*, 45:768–778, 1997.

[63] S. Martello and P. Toth. An exact algorithm for the two–constraint 0–1 knapsack problem. *Operations Research*, 51:826–835, 2003.

[64] R. C. Merkle and M. E. Hellman. Hiding information and signatures in trap door knapsacks. *IEEE Transactions on Information Theory*, IT–24:525–536, 1978.

[65] S. Michel, N. Perrot, and F. Vanderbeck. Knapsack problems with setups. *European Journal of Operational Research*, 196:909–918, 2009.

[66] C. Moore and S. Mertens. *The Nature of Computation*. Oxford University Press, 2011.

[67] H. Müller–Merbach. An improved upper bound for the zero–one knapsack problem: A note on the paper by Martello and Toth. *European Journal of Operational Research*, 2:212–213, 1978.

[68] K. G. Murty. *Linear and combinatorial programming*. Wiley, 1976.

[69] R. M. Nauss. An efficient algorithm for the 0–1 knapsack problem. *Management Science*, 23:27–31, 1976.

[70] U. Pferschy. Dynamic programming revisited: Improving knapsack algorithms. *Computing*, 63:419–430, 1999.

[71] U. Pferschy, D. Pisinger, and G. J. Woeginger. Simple but efficient approaches for the collapsing knapsack problem. *Discrete Applied Mathematics*, 77:271–280, 1997.

[72] U. Pferschy and R. Scatamacchia. Improved dynamic programming and approximation results for the knapsack problem with setups, 2016. To appear in: *International Transactions in Operational Research*.

[73] U. Pferschy and J. Schauer. Approximation of knapsack problems with conflict and forcing graphs, 2016. To appear in: *Journal of Combinatorial Optimization*, available on Optimization Online: http://www.optimization-online.org/DB_HTML/2014/11/4656.html.

[74] D. Pisinger. Algorithms for knapsack problems, 1995. Ph.D thesis, DIKU, University of Copenhagen, Report 95-1.

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

[76] D. Pisinger. A minimal algorithm for the 0–1 knapsack problem. *Operations Research*, 45:758–767, 1997.

[77] D. Pisinger. Core problems in knapsack algorithms. *Operations Research*, 47:570–575, 1999.

[78] D. Pisinger. Linear time algorithms for knapsack problems with bounded weights. *Journal of Algorithms*, 33:1–14, 1999.

[79] D. Pisinger. Where are the hard knapsack problems? *Computers & Operations Research*, 32:2271–2284, 2005.

[80] D. Pisinger and A. Saidi. Tolerance analysis for 0–1 knapsack problems. *European Journal of Operational Research*, 258:866–876, 2017.

[81] D. Pisinger and P. Toth. Knapsack problems. In D.-Z. Du and P. M. Pardalos (eds), editors, *Handbook of Combinatorial Optimization*, volume 1. Kluwer Academic Publisher, 1999.

[82] M. E. Posner and M. Guignard. The collapsing 0–1 knapsack problem. *Mathematical Programming*, 15:155–161, 1978.

[83] K. Pruhs and G. J. Woeginger. Approximation schemes for a class of subset selection problems. *Theoretical Computer Science*, 382:151–156, 2007.

[84] S. Sahni. Approximate algorithms for the 0–1 knapsack problem. *Journal of the ACM*, 22:115–124, 1975.

[85] G. J. Woeginger. Exact algorithms for np–hard problems: a survey. *Lecture Notes in Computer Science*, 2570:185–207, 2003.

[86] L. A. Wolsey. Faces for a linear inequality in 0–1 variables. *Mathematical Programming*, 8:165–178, 1975.

[87] L. A. Wolsey. *Integer Programming*. Wiley, 1998.

[88] J. Wu and T. Srikanthan. An efficient algorithm for the collapsing knapsack problem. *Information Sciences*, 176:1739–1751, 2006.

[89] Y. Yang and R. L. Bulfin. An exact algorithm for the knapsack problem with setup. *International Journal of Operational Research*, 5:280–291, 2009.