VU Research Portal

Column generation strategies and decomposition approaches for the two-stage stochastic multiple knapsack problem

Tönissen, D. D.; van den Akker, J. M.; Hoogeveen, J. A.

Link to publication in VU Research Portal

Column generation strategies and decomposition approaches for the two-stage stochastic multiple knapsack problem

D.D. Tönissen [a,1,*], J.M. van den Akker [b], J.A. Hoogeveen [b]

[a] *School of Industrial Engineering, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands*
[b] *Department of Information and Computing Sciences, Utrecht University, Princetonplein 5, 3584 CC Utrecht, The Netherlands*

A B S T R A C T

Many problems can be formulated by variants of knapsack problems. However, such models are deterministic, while many real-life problems include some kind of uncertainty. Therefore, it is worthwhile to develop and test knapsack models that can deal with disturbances. In this paper, we consider a two-stage stochastic multiple knapsack problem. Here, we have a multiple knapsack problem together with a set of possible disturbances. For each disturbance, or scenario, we know its probability of occurrence and the resulting reduction in the sizes of the knapsacks. For each knapsack we decide in the first stage which items we take with us, and when a disturbance occurs we are allowed to remove items from the corresponding knapsack. Our goal is to find a solution where the expected revenue is maximized. We use branch-and-price to solve this problem. We present and compare two solution approaches: the separate recovery decomposition (SRD) and the combined recovery decomposition (CRD). We prove that the LP-relaxation of the CRD is stronger than the LP-relaxation of the SRD. Furthermore, we investigate numerous column generation strategies and methods to create additional columns outside the pricing problem. These strategies reduce the solution time significantly. To the best of our knowledge, there is no other paper that investigates such strategies so thoroughly.

© 2017 Elsevier Ltd. All rights reserved.

1. Introduction

In this paper we consider a two-stage stochastic multiple knapsack problem (SMKP). The SMKP is a variant of the multiple knapsack problem, where we try to capture some of the uncertainties of the real world. For the SMKP, we have initial (first stage) knapsack sizes, and we have a discrete set of scenarios where the knapsack sizes decrease. Each scenario occurs with a given probability, and we restrict our second stage recovery decision to the removal of items. Consequently, no new items can be added to the knapsack, and we are not allowed to exchange items.

As an example, consider the following situation. There are m workers who perform jobs for clients. These clients issue n requests all together; for each request j ($j = 1, \ldots, n$), we know the time a_j it takes to perform the job and the reward c_j, which is only paid if the job has been fully executed. We further assume that the duration of the job and the size of the reward do not depend on the worker who carries out the job. We know for each worker i

($i = 1, \ldots, m$) the amount of time b_i that he/she can work during the day in normal circumstances. The obvious goal is to find a feasible plan that maximizes the total reward; hence, for each job, we have to determine whether we will accept it and, if accepted, who will do the job. We assume that the clients have to be informed beforehand whether their request is accepted.

The problem sketched above is a typical example of the standard multiple knapsack problem (Kellerer et al. (2004)), which is $\mathcal{NP}$-hard in the strong sense (Li'ang and Suyun 1986) when the number of knapsacks m is part of the input. There is a complication, however, in the form of a small probability that during the day one or multiple workers may get a message that he/she has to leave early to attend some other, more urgent business. This reduces the available work time for a worker i who leaves early to $\bar{b}_i < b_i$ time units. Since each such probability is rather small, it is no option to solve the problem on the basis of the available work time of $\bar{b}_i$ time units. Therefore, we settle for a solution in which each person works at most b_i units of time, but in case one or multiple workers have to leave early, the planned jobs that cannot be executed are cancelled. Hence, next to constructing a solution, we determine beforehand what to do in case of a disturbance. Instead of maximizing the total reward, we then maximize the total expected reward.

---

* Corresponding author.
 E-mail addresses: d.d.tonissen@tue.nl (D.D. Tönissen), J.M.vandenAkker@uu.nl (J.M. van den Akker), J.A.Hoogeveen@uu.nl (J.A. Hoogeveen).
 [1] The research was performed while this author was at Utrecht University.

The SMKP is defined as a multiple knapsack problem together with a discrete scenario set S, which corresponds to the possible disturbances. There are n items, and a_j and c_j denote the size and value of item j, for $j = 1, \ldots, n$. There are m knapsacks; the standard size of knapsack i is equal to b_i $(i = 1, \ldots, m)$. For each scenario $s \in S$, we denote the corresponding size of knapsack i by b_i^s $(i = 1, \ldots, m)$, and we assume that $b_i^s \le b_i$. The probability that scenario $s \in S$ occurs is equal to p_s and we use p_0 to denote the probability that there are no disturbances. We only allow recovery by removing items. Our goal is to find a solution such that the expected value is maximum.

This paper extends the two-stage stochastic single knapsack problem introduced in van den Akker et al. (2016) (which itself is an extension to Bouman, 2011) to multiple knapsacks. We use the two decomposition approaches introduced in van den Akker et al. (2016): the separate recovery decomposition (SRD), and the combined recovery decomposition (CRD). In the SRD formulation the variables correspond to independent knapsack fillings for the undisturbed situation and independent knapsack fillings for all of the scenarios; one knapsack filling has to be selected for the undisturbed situation and one knapsack filling for each scenario. The constraint that the knapsack fillings of the scenarios have to be a subset of the knapsack filling for the undisturbed problem (it is only allowed to remove items) is enforced in the master problem. In the CRD formulation we have for each scenario a set of variables that correspond to a combination of a knapsack filling for the undisturbed situation together with the optimal knapsack filling for the scenario that is compatible with the knapsack filling for the undisturbed situation; the subset constraint is now directly satisfied within the variables of the problem. Constraints are included in the master problem to enforce the use of the same initial knapsack filling in each scenario. The computational experiments indicate that the SRD formulation performs better than the CRD formulation for the two-stage stochastic single knapsack problem. van den Akker et al. (2016) also study the demand robust shortest path problem, which is a variant of the shortest path problem in which the sink is unknown and the edges become more expensive after the sink has been revealed. For the demand robust shortest path problem the CRD formulation performs really well. The computational experiments for the demand robust shortest path problem indicate the importance of finding a good approach for generating and adding columns to the master problem. This, in combination with generating additional columns in a smart way, reduces the solution time usually by a factor of at least 10, and in some examples, even by a factor of more than 50.

This paper provides an extensive computational study of column generation and column addition strategies. We investigate the differences between the CRD and SRD formulations for the SMKP in great detail. Finding the best (or at least a good) column generation and decomposition strategy is very important as it significantly reduces the solution time. Furthermore, this paper is the first paper about the two-stage stochastic multiple knapsack problem.

The paper is organized as follows. In Section 2 we perform a literature review. In Section 3 we present the decomposition approaches, and we prove that the CRD formulation has a stronger LP-relaxation than the SRD formulation. In Section 4 we demonstrate our method to generate good test instances for the SMKP. In Section 5 we present and test our column generation approaches and show that using a smarter approach can significantly decrease the solution time (with a factor of ~ 10) compared to naive approaches. We also study the influence of the number of knapsacks, items and scenarios of the problem. In Section 6 we present our branch-and-price algorithm, and in Section 7 we compare the performance of the SRD and CRD formulations. In the final section, we summarize our conclusions and present ideas for future research.

2. Literature review

The multiple knapsack problem without disturbances can be solved through dynamic programming in $\mathcal{O}(n \cdot b_{\max}^m)$ time, where $b_{\max}$ is the maximum size of all knapsacks. In the literature, an exact solution of this problem is often found by variants of branch-and-bound (Martello and Toth 1980) or bound-and-bound (Martello and Toth, 1981; Pisinger, 1999) algorithms, where either a Lagrangian or surrogate relaxation bound is used. In a bound-and-bound algorithm, the maximization problem uses a lower bound besides an upper bound to determine which branches to follow in the decision tree. A slightly different approach is found in Fukunaga (2011), which integrates the bound-and-bound mechanism with a bin-orientated approach, using path-symmetry and path-dominance for pruning nodes.

Although we use an average case objective, our model and solution approach are inspired by the recoverable robustness literature. Recoverable robustness was introduced by Liebchen et al. (2009) for railway optimization, where it has gained a lot of attention since then (see for example Cacchiani et al., 2008; Cicerone et al., 2009). Recoverable robustness can be seen as two-stage robust optimization, with the additional restriction that it uses a pre-described, fast and simple recovery algorithm to make a solution feasible for a set of given scenarios. This restriction makes recoverable robustness very suitable for combinatorial problems.

We are aware of two papers (Büsing et al., 2011a; 2011b) that study the knapsack problem within the recoverable robustness framework. In Büsing et al. (2011b), the authors solve admission control problems on a single link of a telecommunication network with the help of a recoverable robust knapsack problem. The authors consider a single knapsack problem, which has uncertainty in the sizes and the revenues of the items. The recovery consists of adding at most l items and removing at most k items. The value of k is determined as a fraction of the number of items included in the knapsack; similarly, l depends on the number of items that are not in the knapsack. The authors study the gain of recovery by varying these fractions between 0 and 100%; allowing recovery yields a gain up to 45%. Furthermore, the authors show that this problem is weakly $\mathcal{NP}$-hard for a fixed number of scenarios, and that the problem becomes strongly $\mathcal{NP}$-hard when the number of scenarios is part of the input. A follow-up paper Büsing et al. (2011a) presents an integer linear programming formulation of quadratic size and evaluates the gain of recovery.

There is a lot of literature on the stochastic knapsack problem. Most of the literature discusses the single knapsack problem with random item sizes. There are two ways to deal with a possible overload. When an overload is acceptable, as long as its probability of occurrence is within a certain bound, then the knapsack constraint can be replaced by a chance constraint (Goel and Indyk, 1999; Kleinberg et al., 2000; Kosuch and Lisser, 2010). When overflow is not acceptable the last inserted item is removed from the knapsack (Bhalgat and Khanna, 2011; Dean et al., 2008) (in this case the items are added to the knapsack one by one and the size is revealed when the item is added to the knapsack), the knapsack returns zero when it overflows (Chen and Ross 2014), or a penalty is incurred (Kleywegt and Papastavrou 2001). In other papers, the profits are uncertain, and the objective is to find a set of items that maximizes the probability to achieve some target reward (Carraway et al., 1993; Henig, 1990; Morton and Wood, 1998).

Kosuch and Lisser (2011), and Kosuch (2014), both study a two-stage stochastic knapsack problem with random item sizes. In Kosuch and Lisser (2011), the item sizes are normally distributed, and recovery is limited to the addition and removal of items. A chance constraint is used in the first stage to restrict the probability of an overload in the second stage. Because the sizes are assumed to be normally distributed, no method is known to

exactly evaluate the expectation of the second-stage solution for a given first-stage decision. Instead, a method is proposed to compute lower bounds, and a branch-and-bound framework is used to find the first-stage solutions that provide the best lower bounds. To calculate relative gaps, an upper bound is computed by solving a continuous version of the problem with a stochastic gradient algorithm. In Kosuch (2014) the items are discretely distributed, items can also be exchanged and approximation algorithms are given for special cases. Furthermore, the authors give a non-approximation result: the problem cannot be approximated in polynomial time with a better ratio than $K^{1/2}$ (where K is the number of second-stage scenarios). Gaivoronski et al. (2011) study a quadratic two-stage stochastic knapsack problem with random item sizes and revenues, where (a part of) the information is revealed in the second stage. The seconds stage allows for a recourse decision and a chance constraint is used on the capacity of the knapsack in the second stage.

To the best of our knowledge there is no other literature that studies the two-stage stochastic multiple knapsack problem. Closely related is the work of van den Akker et al. (2016), which studies the single knapsack version of this problem. Besides the decomposition approaches and branch-and-price algorithm (see Section 1), van den Akker et al. (2016) test several other algorithms on the two-stage stochastic knapsack problem. These algorithms include branch-and-bound, dynamic programming and local search. The computational experiments indicate that the SRD and some of the local search algorithms perform well. The dynamic programming algorithm performs very poorly, while the CRD also has poor performance. Branch-and-bound performs better than the CRD, but the SRD outperforms branch-and-bound. We believe that the CRD performs poorly because for the two-stage stochastic knapsack problem the pricing problem of the CRD is significantly more difficult than that of the SRD.

3. The decomposition approaches

In this section, the SRD and CRD formulations for the two-stage stochastic multiple knapsack problem are presented, analyzed, and compared. For both formulations, integer linear programs are provided and we show how to solve them using a branch-and-price algorithm. Furthermore, we compare the two approaches theoretically. Let us start with the SRD formulation for the ease of explanation.

3.1. The separate recovery decomposition formulation

When there is only one knapsack (see van den Akker et al., 2016), one initial knapsack filling and one knapsack filling for each scenario have to be selected in the SRD formulation. The constraint that the knapsack fillings of the scenarios have to be a subset of the initial filling (we are only allowed to remove items) is enforced in the master problem. When there are more knapsacks, we can apply a similar formulation, because we can describe any feasible solution of the SMKP by combining one initial multiple knapsack filling and a multiple knapsack filling for each scenario. Since the multiple knapsack problem cannot be solved in polynomial or pseudo-polynomial time we apply one more decomposition by considering the m knapsacks individually. The feasible solutions are now found by combining m initial knapsack fillings and m knapsack fillings for each scenario. Thus, we need a total of $(|S| + 1) \cdot m$ knapsack fillings, where m is the number of knapsacks and $|S|$ the number of scenarios.

In our integer linear program, we work with binary variables that indicate whether we use a given knapsack filling for a given knapsack $i (i = 1, \ldots, m)$ for a specific scenario $s \in S$. For each

knapsack i, we define K_i as the set of all feasible undisturbed knapsack fillings; K_i^s is defined similarly for each scenario $s \in S$.

We use the following parameters to characterize a knapsack filling:

$$a_{ijk} = \begin{cases} 1 & \text{if item } j \text{ belongs to knapsack filling } k \in K_i, \\ 0 & \text{otherwise}; \end{cases}$$

$$a_{ijq}^s = \begin{cases} 1 & \text{if item } j \text{ belongs to knapsack filling } q \in K_i^s, \\ 0 & \text{otherwise}. \end{cases}$$

We define two types of decision variables:

$$v_{ik} = \begin{cases} 1 & \text{if knapsack filling } k \in K_i \text{ is selected} \\ 0 & \text{otherwise}; \end{cases}$$

$$y_{iq}^s = \begin{cases} 1 & \text{if knapsack filling } q \in K_i^s \text{ is selected}, \\ 0 & \text{otherwise}. \end{cases}$$

Obviously, we only introduce variables v_{ik} and y_{iq}^s if $k \in K_i$ and $q \in K_i^s$, respectively.

We define C_{ik} as the reward of the items in the knapsack filling k for the initial solution of knapsack i; C_{iq}^s is defined similarly for recovery knapsack filling q and scenario s. For ease of notation, we use $M = \{1, \ldots, m\}$ for the knapsacks and $N = \{1, \ldots, n\}$ for the items. The separate recovery decomposition formulation for the SMKP is now

$$\max p_0 \sum_{i \in M} \sum_{k \in K_i} C_{ik} v_{ik} + \sum_{s \in S} p_s \sum_{i \in M} \sum_{q \in K_i^s} C_{iq}^s y_{iq}^s$$

subject to

$$\sum_{k \in K_i} v_{ik} = 1 \qquad \forall i \in M, \tag{1}$$

$$\sum_{q \in K_i^s} y_{iq}^s = 1 \qquad \forall i \in M, \quad s \in S, \tag{2}$$

$$\sum_{k \in K_i} a_{ijk} v_{ik} - \sum_{q \in K_i^s} a_{ijq}^s y_{iq}^s \geq 0 \quad \forall i \in M, \quad j \in N, \quad s \in S, \tag{3}$$

$$\sum_{i \in M} \sum_{k \in K_i} a_{ijk} v_{ik} \leq 1 \qquad \forall j \in N, \tag{4}$$

$$v_{ik} \in \{0, 1\} \qquad \forall i \in M, \quad k \in K_i, \tag{5}$$

$$y_{iq}^s \in \{0, 1\} \qquad \forall i \in M, \quad s \in S, \quad k \in K_i^s. \tag{6}$$

Constraints (1) ensure that exactly one filling is selected for every knapsack for the undisturbed situation, and Constraints (2) ensure that exactly one knapsack filling is selected for every recovery situation. Constraints (3) guarantee that recovery is done by removing items, and Constraints (4) ensure that every item is in at most one selected initial knapsack.

We relax the integrality Constraints (5) and (6) and use branch-and-price Barnhart et al. (1998) to find the integral optimum. The value of the maximization objective increases if and only if the reduced costs are positive. Let λ_i, μ_{is}, π_{ijs} and ρ_j be the dual variables of Constraints (1) to (4). The reduced cost of the variable v_{ik}, which we denote as $c^{red}(v_{ik})$, is then equal to

$$c^{red}(v_{ik}) = p_0 C_{ik} - \lambda_i - \sum_{j \in N} \sum_{s \in S} a_{ijk} \pi_{ijs} - \sum_{j \in N} a_{ijk} \rho_j$$

$$= \sum_{j \in N} a_{ijk} \left(p_0 c_j - \sum_{s \in S} \pi_{ijs} - \rho_j \right) - \lambda_i,$$

since $C_{ik} = \sum_{j \in N} c_j a_{ijk}$.

To find a variable v_{ik} with positive reduced cost, if one exists, we maximize the above expression for the reduced cost by selecting the optimal values for a_{ijk} subject to the constraint that the resulting filling is feasible for the given knapsack i ($i = 1, \ldots, m$). This results in a knapsack problem where the revenue of item j equals $p_0 c_j - \sum_{s \in S} \pi_{ijs} - \rho_j$, and the size of the knapsack equals b_i.

Similarly the reduced cost of the variable y_{iq}^s is given by

$$c^{red}(y_{iq}^s) = p_s C_{iq}^s - \mu_{is} + \sum_{j \in N} a_{ijq}^s \pi_{ijs} = \sum_{j \in N} a_{ijk}(p_s c_j + \pi_{ijs}) - \mu_{is}.$$

Again, if we want to find a variable y_{iq}^s with maximum reduced cost for a given knapsack i ($i = 1, \ldots, m$) and scenario s ($s \in S$), then we have to solve a knapsack problem. Here the revenue of item j equals $p_s c_j + \pi_{ijs}$, and the size of the knapsack equals b_i^s.

In both cases the pricing problem is a knapsack problem, which can be solved in pseudo-polynomial time by dynamic programming. We use a preprocessing step to remove items with negative revenue or with a size larger than the knapsack size before starting the algorithm. The complexity of the implemented algorithm is $\mathcal{O}(\min(2^n, n \cdot b_i))$. There are at most 2^n recursive calls, which is the first part of the bound. Furthermore, the complexity of the standard dynamic programming algorithm for the knapsack problem is $\mathcal{O}(n \cdot b_i)$, as it iterates from 1 to n and from 0 to b_i. Our implementation only uses the knapsack sizes which actually can be reached with the chosen items and can skip steps between 0 and b_i. Consequently, it cannot use more than $\mathcal{O}(n \cdot b_i)$ steps, which is the second part of the bound.

3.2. The combined recovery decomposition formulation

When there is only one knapsack (see van den Akker et al., 2016), a combination of an initial knapsack filling and the best scenario knapsack filling given the initial knapsack filling is selected for each scenario in the CRD formulation. The subset constraint is now directly satisfied within the columns of the problem. We include constraints in the master problem to enforce the use of the same initial knapsack filling in each scenario. Because the multiple knapsack problem can not be solved in polynomial or pseudo-polynomial time we again apply an additional decomposition by considering the m knapsacks individually. Any feasible solution to the SRMKP can now be described as combining an initial knapsack filling for knapsack i and the best knapsack filling for that scenario given the initial knapsack filling for each scenario. This gives a total of $|S| \cdot m$ combined (initial/recovery) knapsack fillings.

We use two types of variables in our formulation. The first type indicates whether item j ($j = 1, \ldots, n$) is included in the initial filling of knapsack i ($i = 1, \ldots, m$). Therefore, we define

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is contained in knapsack } i, \\ 0 & \text{otherwise.} \end{cases}$$

The second type of variables indicate whether a combined knapsack filling consisting of an initial and recovery filling for a specific knapsack i ($i = 1, \ldots, m$) and scenario s ($s \in S$) is part of the solution.

$$z_{kqi}^s = \begin{cases} 1 & \text{if the combination of initial knapsack } k \text{ and} \\ & \text{recovery knapsack } q \text{ for scenario } s, \\ & \text{is selected for knapsack } i, \\ 0 & \text{otherwise.} \end{cases}$$

Obviously, we introduce a variable z_{kqi}^s only if it corresponds to a feasible combined knapsack filling. We define KQ_i^s as the set containing all possible, feasible combinations (k, q) of an initial filling k and a recovery q for knapsack i ($i = 1, \ldots, m$) and scenario s ($s \in S$). As before, we use the parameters a_{ijk} to indicate whether item j belongs to knapsack filling k of knapsack i.

We formulate the problem as follows:

$$\max p_0 \sum_{i \in M} \sum_{j \in N} c_j x_{ij} + \sum_{s \in S} p_s \sum_{i \in M} \sum_{(k,q) \in KQ_i^s} C_{iq}^s z_{kqi}^s$$

subject to

$$\sum_{(k,q) \in KQ_i^s} z_{kqi}^s = 1 \quad \forall i \in M, \quad s \in S, \tag{7}$$

$$x_{ij} - \sum_{(k,q) \in KQ_i^s} a_{ijk} z_{kqi}^s = 0 \quad \forall i \in M, \quad j \in N, \quad s \in S, \tag{8}$$

$$\sum_{i \in M} x_{ij} \leq 1 \quad \forall j \in N, \tag{9}$$

$$x_{ij} \in \{0, 1\} \quad \forall i \in M, \quad j \in N, \tag{10}$$

$$z_{kqi}^s \in \{0, 1\} \quad \forall (k, q) \in KQ_i^s, \quad i \in M, \quad s \in S. \tag{11}$$

Constraints (7) enforce that exactly one combination is selected for each knapsack for every scenario. Constraints (8) ensure that in each scenario we have the same initial filling for knapsack i ($i = 1, \ldots, m$), whereas Constraints (9) ensure that every item is in at most one of the m knapsacks.

We use an LP-relaxation on the variables x_{ij} and z_{kqi}^s and solve the integer problem with branch-and-price, just as the SRD formulation. Pricing is performed for every knapsack and scenario combination. We denote the dual variables of Constraints (7) and (8) as ρ_{is} and σ_{ijs}. The reduced cost of a variable z_{kqi}^s for a given knapsack i and scenario s is then equal to

$$c^{red}(z_{kqi}^s) = C_{iq}^s + \sum_{j=1}^n a_{ijk} \sigma_{ijs} - \rho_{is} = \sum_{j=1}^n a_{ijq} p_s c_j + \sum_{j=1}^n a_{ijk} \sigma_{ijs} - \rho_{is}.$$

In each iteration, we have a pricing problem for every scenario s and knapsack i. This gives us a total of $m \cdot |S|$ combined (initial/recovery) knapsack fillings, among which we have to choose at least one in every iteration. For each combined knapsack problem, we have two possibilities: the scenario knapsack size decreases ($b_i^s < b_i$) or it stays the same ($b_i = b_i^s$). When it does not change, we use the same $\mathcal{O}(\min(n \cdot b_i, 2^n))$ dynamic programming algorithm as for the SRD formulation, but in this case the revenue of item j is $\sigma_{ijs} + p_s c_j$, because the found knapsack filling is used for both the initial and the recovery part of the column. If the scenario knapsack size decreases, the initial and recovery situations can become different. Because the recovery consists of removing items, we have three options for each item: We can take the item initially and as recovery, we can take it only initially, and we can exclude it from the knapsack. The revenue of taking an item j both initially and as recovery is $\sigma_{ijs} + p_s c_j$ and for taking it initially is σ_{ijs}. Consequently, the dynamic programming algorithm has complexity $\mathcal{O}(\min(n \cdot b_i^s \cdot b_i, 3^n))$.

To increase the solution speed, we use preprocessing to eliminate as many items as possible. We remove items when the weight of the item is larger than the initial knapsack size ($a_j > b_i$), when the revenue is always negative ($\sigma_{ijs} + p_s c_j < 0$), and when the weight of the item is larger than the scenario knapsack size and the revenue of taking the item initially is negative ($a_j > b_i^s$ and $\sigma_{ijs} < 0$).

3.3. Theoretical comparison

Before doing computational experiments, it is important to investigate theoretical properties of the formulations. Table 1 shows the main characteristics of the formulations, where m, n and $|S|$ are the number of knapsacks, items and scenarios.

Table 1
Comparing the decompositions.

	SRD	CRD
Number of constraints	$m + (m \cdot |S|) + (m \cdot n \cdot |S|) + n$	$(m \cdot |S|) + (m \cdot n \cdot |S|) + n$
Number of pricing problems	$m + (m \cdot |S|)$	$m \cdot |S|$
Pricing problem complexity	$\mathcal{O}(\min(n \cdot b_i, 2^n))$	$\mathcal{O}(\min(n \cdot b_i, 2^n))$ or $\mathcal{O}(\min(n \cdot b_i^s \cdot b_i, 3^n))$

The CRD formulation has m constraints and m possible pricing problems fewer than the SRD formulation. However, the complexity of generating a column for the CRD formulation is higher than for the SRD formulation when the column corresponds to a scenario in which the size of the knapsack decreases. We expect that the last characteristic is the reason why the SRD formulation performs better for $m = 1$ in van den Akker et al. (2016). Furthermore, for $m = 1$ the knapsack size always decreases and consequently the pricing problem is always more difficult. For $m \geq 2$ not all knapsack sizes have to decrease, in which case the pricing problem complexity is only $\mathcal{O}(\min(n \cdot b_i, 2^n))$. We can prove that the LP-relaxation of the CRD formulation is stronger than that of the SRD formulation. This theorem is similar to a theorem presented in van den Akker et al. (2016), but does not directly follow from this theorem, since we also apply a decomposition based on the m knapsacks. The theorem is in favor of the CRD formulation, because a strong upper bound generally prunes more nodes. We define Z_{LP}^{SRD} and Z_{LP}^{CRD} as the optimal solution values of the LP-relaxations of the SRD and CRD formulations, respectively.

Theorem 1. $Z_{LP}^{SRD} \geq Z_{LP}^{CRD}$, and there are instances for which the inequality is strict.

Proof. We prove $Z_{LP}^{SRD} \geq Z_{LP}^{CRD}$ by showing that for each given, feasible solution for the LP-relaxation of the CRD formulation, we can find a solution for the SRD formulation with the same revenue. Furthermore, we show by example that there is at least one instance for which $Z_{LP}^{SRD} > Z_{LP}^{CRD}$.

Assume that we have a feasible solution for the LP-relaxation value of the CRD formulation; this solution is characterized by the values for x_{ij} and z_{kqi}^s.

Before indicating how to find the corresponding solution of the SRD formulation, we first remark that from Constraint (8) it follows that

$$x_{ij} = \sum_{(k,q) \in KQ_i^1} a_{ijk} z_{kqi}^1,$$

since the constraint holds for all $s \in S$. We will specify the solution of the SDR formulation on the basis of z_{kqi}^1 values, but choosing any other scenario will lead to the same solution. We use this expression to find the value v_{ik} in the corresponding solution, which indicates how much of knapsack filling $k \in K_i$ is used for knapsack i in the initial solution of the SRD formulation. In the CRD formulation filling k is used in the initial solution for knapsack i in combination with all feasible recoveries q for which $(k, q) \in KQ_i^1$, and such a combination is chosen with value z_{kqi}^1. Hence, summing over the relevant fillings q, we find

$$v_{ik} = \sum_{q:(k,q) \in KQ_i^1} z_{kqi}^1,$$

but we would have found the same value for any scenario $s \in S$ instead of scenario 1. Next, we consider y_{iq}^s, which in the SRD formulation indicates how much of knapsack filling q we use for knapsack i as a recovery for scenario s. In the CRD formulation, we use q in combination with an initial filling k; hence, we must sum over

Table 2
Items for the example.

Item	Weight	Revenue
Item 1	2	9
Item 2	1	5
Item 3	1	5

all possible initial fillings k such that $(k, q) \in KQ_i^s$. This leads us to the choice of

$$y_{iq}^s = \sum_{k:(k,q) \in KQ_i^s} z_{kqi}^s.$$

Since the choice of these values for the SRD formulation describes the same solution as the CRD formulation, we know that this solution of the SRD formulation is feasible. A similar statement could be made with respect to the value of the objective value, but for the sake of completeness we will prove it below. We first prove that the first part of the value of the objective function is the same for the SRD and CRD formulation. We have that

$$p_0 \sum_{i \in M} \sum_{k \in K_i} C_{ik} v_{ik} = p_0 \sum_{i \in M} \sum_{k \in K_i} C_{ik} \sum_{q:(k,q) \in KQ_i^1} z_{kqi}^1.$$

Using that $\sum_{k \in K_i} \sum_{q:(k,q) \in KQ_i^1} = \sum_{(k,q) \in KQ_i^1}$, and $C_{ik} = \sum_{j \in N} c_j a_{ijk}$, we find that the last expression is equal to

$$p_0 \sum_{i \in M} \sum_{j \in N} \sum_{(k,q) \in KQ_i^1} c_j a_{ijk} z_{kqi}^1 = p_0 \sum_{i \in M} \sum_{j \in N} c_j x_{ij},$$

which is exactly the first part of the objective value of the CRD. Using similar arguments we find that

$$\sum_{s \in S} p_s \sum_{i \in M} \sum_{q \in K_i^s} C_{iq}^s y_{iq}^s = \sum_{s \in S} p_s \sum_{i \in M} \sum_{q \in K_i^s} \sum_{k:(k,q) \in KQ_i^s} C_{iq}^s z_{kqi}^s =$$

$$\sum_{s \in S} p_s \sum_{i \in M} \sum_{(k,q) \in KQ_i^s} C_{iq}^s z_{kqi}^s,$$

which shows that the second part of the values of the objective functions of the CDR and SDR formulations are the same.

Furthermore, we can illustrate by an example that for some instances the objective value of the SRD formulation is larger. This example has one knapsack with size 3 and one scenario, in which the size of the knapsack decreases to 2 with a probability of 0.3. We have three items and their weights and rewards can be found in Table 2.

An optimal solution of the CRD formulation is:

- initial {1, 2} with recovery {1}.
- optimal objective is $0.7 \cdot 14 + 0.3 \cdot 9 = 12.5$.

For the SRD formulation the following solution is feasible:

- Initial: 0.5 times {1, 2} and 0.5 times {1, 3}
- Recovery: 0.5 times {1} and 0.5 times {2, 3}
- optimal objective is $0.7(0.5 \cdot 14 + 0.5 \cdot 14) + 0.3(0.5 \cdot 9 + 0.5 \cdot 10) = 12.65$.

The solution of the SRD formulation is not feasible for the CRD formulation, as every recovery has to contain a subset of the items of one initial situation. Therefore, the CRD formulation cannot combine initial solutions, whereas the SRD formulation is allowed to use combined recoveries. By using recoveries that are a subset of more than one initial solution, it is possible that the SRD formulation gives a solution with a larger solution value.

We can conclude that the LP-relaxation of the CRD formulation is stronger than that of the SRD formulation, which implies that its value is closer to the optimal integral solution. □

4. Generating random test instances

When doing extensive computational research, it is important to have interesting instances for the two-stage stochastic multiple knapsack problem. In this section we describe our technique to generate good instances. The random data are always drawn from the uniform distribution, thus we only specify the corresponding interval. Knapsack sizes etc. are integral by default; if fractional, these values are rounded down.

4.1. Generating the items

Pisinger (2004) describes different instance classes of the single knapsack problem. We work with two main instance sets: a diverse set and a more structured instance set. The larger and more diverse instance set consists of all instance classes, while the more structured instance set focusses on the strongly correlated instances. For the strongly correlated instances, the weight a_j is drawn randomly from $[1, R]$ and the revenue is $c_j = a_j + \frac{R}{10}$, where R is a random range parameter.

We consider strongly correlated instances because they correspond to classical benchmark for the (multiple) knapsack problem, and they are hard to solve. According to Pisinger (2004) these instance are hard because of two reasons: *"(a) The instances are ill-conditioned in the sense that there is a large gap between the continuous and integer solution of the problem. (b) Sorting the items according to decreasing efficiencies corresponds to a sorting according to the weights. Thus for any small interval of the ordered items (i.e. a core) there is a limited variation in the weights, making it difficult to satisfy the capacity constraint with equality."*.

4.2. Generating the knapsack sizes

Pisinger (1999) introduces two classes of knapsack sizes. The first class has dissimilar sizes: the knapsack sizes b_i ($i = 1, \ldots, m - 1$) are generated randomly from

$$\left[0, \left(K \sum_{j=1}^{n} a_j - \sum_{k=1}^{i-1} b_k \right) \right] \text{for } i = 1, \ldots, m-1, \tag{12}$$

where K is always set to 0.5. The other class has similar sizes, which are generated randomly in the range

$$\left[0.4 \sum_{j=1}^{n} \frac{a_j}{m}, 0.6 \sum_{j=1}^{n} \frac{a_j}{m} \right] \text{for } i = 1, \ldots, m-1. \tag{13}$$

For both classes, the capacity of the mth class was set to

$$b_m = 0.5 \sum_{j=1}^{n} a_j - \sum_{i=1}^{m-1} b_i. \tag{14}$$

To avoid trivial problems, they have to satisfy the following properties:

1. All items can fit in at least one knapsack.

2. All knapsacks should have a size larger than or equal to the weight of the smallest item.
3. All knapsack sizes should be smaller than the sum of the weights of all items.

Pisinger checks if an instance satisfies these assumptions and if not the instance is removed and a new one is created. We want to do this differently and directly create instances that satisfy these properties. Therefore, we have to modify Eq (12). To satisfy the first property the size of at least one knapsack has to be larger than or equal to the maximum item weight ($a_{\max}$). By property 2, the other knapsack sizes have to be larger than or equal to the minimum item weight ($a_{\min}$). The last property gives an upper bound on the size of the knapsack $\sum_{j=1}^{n} a_j$.

We want to focus on dissimilar instances. To generate these, we modify Pisinger's method, to create more diversity. Instead of setting $K = 0.5$, we generate K randomly from

$$\left[\frac{a_{\max} + (m-1)a_{\min}}{\sum_{j=1}^{n} a_j} + 0.1, 1.0 \right). \tag{15}$$

Note that $a_{\max} + (m-1)a_{\min}$ in Eq. (15), corresponds to the minimal summation over all knapsack sizes, due to properties 1 and 2.

Our first knapsack size b_1 is generated randomly from

$$\left[a_{\max}, \left(K \sum_{j=1}^{n} a_j - (m-1)a_{\min} \right) \right]. \tag{16}$$

The lower bound $a_{\max}$ guarantees that there is at least one knapsack with weight $a_{\max}$ or more. The upper bound is equal to the maximum size of all knapsacks together, from which we subtract the minimum size of the other knapsacks. Consequently, the first knapsack always satisfies properties 1 to 3, and it is always possible to generate $m - 1$ other knapsacks satisfying these properties.

The other knapsack sizes are generated randomly from

$$\left[a_{\min}, \left(K \sum_{j=1}^{n} a_j - \left(\sum_{k=1}^{i-1} b_k + (m-i)a_{\min} \right) \right) \right] \text{for } i = 2, \ldots, m. \tag{17}$$

The lower bound ensures that the sizes of those knapsacks are at least $a_{\min}$, such that property 2 is always satisfied. The upper bound is the maximum allowed size of all knapsacks together, from which we subtract the size of the already generated knapsacks and the minimum size of the knapsacks that still have to be generated. Again, the knapsack size always satisfies the properties, and it is always possible to generate $m - i$ other knapsacks that satisfy these properties.

Experiments show that this method works well only when there are a few knapsacks. When there are four or more knapsacks, the first knapsack size is large, and the other knapsack sizes are very small. Therefore, we improve the method by generating the knapsack sizes in sets of three. We define $W = \frac{\sum_{j=1}^{n} a_j}{\lceil \frac{m}{3} \rceil}$, which is the maximum weight of each set.

The last set can have fewer than three knapsacks and in that case, the weight is divided between those knapsacks. They can become slightly larger than the knapsacks of the other groups, but this is not a problem because we want to generate diverse knapsack sizes.

K is now generated randomly from

$$\left[\frac{a_{\max} + d\ a_{\min}}{W} + 0.1, 1.0 \right) \text{with } d = 3 \text{ if } m \geq 3 \text{ else } d = m, \tag{18}$$

and the first knapsack has a size generated randomly from

$$[a_{\max}, (K \cdot W - (d-1)a_{\min})]. \tag{19}$$

We denote r as the number of sets made so far. We generate sets of $d = \min\{3, m - 1 - (3 \cdot r)\}$ knapsacks, until m knapsacks have been generated randomly from

$$b_{i+1+3r} = \left[a_{\min}, \left(K \cdot W - \left(\sum_{k=1}^{i-1} b_k + (d-i)a_{\min} \right) \right) \right] \text{ for } i = 1, \ldots, d. \tag{20}$$

4.3. Generating the scenarios

For every generated initial instance (n, m, c_j, a_j ($j = 1, \ldots, n$), b_j ($i = 1, \ldots, m$)) we generate one set of scenarios. A scenario is characterized by the probability it occurs and the amounts by which one or more knapsacks decrease. We assume that only a subset of knapsacks is prone to disturbances. There are uncertain knapsacks, which can decrease in size in one or more scenarios, and fixed knapsacks, which will never decrease in size. The first step in generating the scenarios is to generate the set of uncertain knapsacks. The size of each knapsack is uncertain with probability D. When $D = 1$ all knapsacks are uncertain and when $D = 0$ we have the classical multiple knapsack problem. We avoid $D = 0$, and when the generated set of uncertain knapsacks happens to be empty, we randomly select one knapsack and add it to the set. This guarantees that we always have at least one scenario.

Even for uncertain knapsacks, there is a probability that their size will not decrease. The latter is generated randomly from [0.1, 0.9]. For uncertain knapsacks, different decreases are possible and this number is generated randomly from [1, max], where max is a parameter. Every decrease has a relative probability of occurrence, which is uniformly generated in [1, r] (r is a parameter) and then normalized. For every possible decrease its amount is given by a fraction of the original knapsack size. Hence, the new size of the knapsack is $f \cdot b_i$, where f is generated randomly from [u, 1.0], where u is a parameter. For every possible combination of knapsack size decreases (including no decrease) of the different knapsacks, a scenario is generated. The probability of each scenario is computed by multiplying the probabilities of the corresponding events for the separate knapsacks.

We demonstrate this with an example. Assume that we want to generate an instance with 3 knapsacks, where $D = 0.5$ and $max = 3$. We first determine which knapsacks are uncertain; suppose that there are two such knapsacks, which we denote by knapsacks 1 and 2. Now we generate the probabilities that they do not change randomly from [0.1, 0.9]; the outcome is 0.7 for knapsack 1 and 0.4 for knapsack 2. The next step is to generate for both knapsacks the number of possible decreases, which is generated randomly from [1, 3]. The outcome is that knapsack 1 has one decrease and that knapsack 2 has two possible decreases, which we denote by A and B. Subsequently, we generate the relative probability that decrease A and decrease B occurs for knapsack 2. These relative probabilities are 0.2 for A and 0.8 for B. We further generate that knapsack 1 has initial size 20, and that it can decrease to 18; knapsack 2 has initial size 28, and it can decrease to 25 (A) and 18 (B), respectively. Knapsack 3 has size 30, which stays the same for every possible scenario.

We translate this into a set of scenarios by enumerating all possibilities and computing the respective probabilities. We find the following set:

1. Knapsack 1 decreases in size from 20 to 18, and the other knapsacks stay the same; $p_1 = 0.3 \cdot 0.4 \cdot 1.0 = 0.12$.
2. Knapsack 2 (case A) decreases in size from 28 to 25, and the other knapsacks stay the same; $p_2 = 0.7 \cdot 0.6 \cdot 0.2 \cdot 1.0 = 0.084$.
3. Knapsack 2 (case B) decreases in size from 28 to 18, and the other knapsacks stay the same; $p_3 = 0.7 \cdot 0.6 \cdot 0.8 \cdot 1.0 = 0.366$.

Table 3
Instance sets.

Sets	Number of instances	u	max	D	Knapsacks	Items	Scenarios
Small	1000	0.5	2	0.5	[2, 7]	[4,21]	9.8
Large	8400	0.25	2	0.5	[2,8]	[4,39]	9.9
Structured	2859	0.5	3	1.0	[1,12]	[5,25]	38.4

4. Knapsacks 1 and 2 (case A) decrease in size, and knapsack 3 stays the same; $p_4 = 0.3 \cdot 0.6 \cdot 0.2 \cdot 1.0 = 0.036$.
5. Knapsacks 1 and 2 (case B) decrease in size, and knapsack 3 stays the same; $p_5 = 0.3 \cdot 0.6 \cdot 0.8 \cdot 1.0 = 0.144$.

Furthermore, we have the initial situation where the sizes do not change with probability $p_0 = 0.7 \cdot 0.4 \cdot 1.0 = 0.28$.

This approach generates all possible scenarios. If we want to generate a given number of scenarios, we can select that number randomly from the generated scenarios, and normalize the probabilities.

4.4. Our test sets

We generate the item sizes using $R = 30$, and $r = 3$ for all our test sets. Table 3 shows the other parameters with which we generate our test sets, the range for the number of items and knapsacks and the average number of scenarios for each set of instances.

The large instance set is a very diverse large test set that contains 700 instances for each of the instance classes from Pisinger (2004). For the small and large instance sets, all possible scenarios are always generated, but for the structured set, we only select a part of the generated scenarios and normalize the probabilities. The average number of knapsacks and items for the small set are 4.5 and 11.2, and for the large set they are 5.0 and 13.6 respectively. The structured instance set is specifically made to study the influence of the number of knapsacks, items and scenarios on the CRD and SRD formulations. For this reason (and the reason described in Section 4.1), all instances are from the strongly connected item class. The instances of this set are made using the following scheme. We considered each possible number of knapsacks and items in their corresponding range, and we considered instances with 0, 1, 5, 10, 20, 30, ..., 100 scenarios; afterwards, we removed the instances that were meaningless (for example with more knapsacks than items). The instances with only one knapsack are representative of the two-stage knapsack problem, while we have a normal knapsack or multiple knapsack problem when there is no scenario. The instances with no scenario are associated with a dummy scenario in which the knapsack size stays the same. This is necessary to make the CRD formulation possible. The average number of knapsacks and items for the structured set are 6.7 and 15.6 respectively.

All test sets can be found at http://home.ieis.tue.nl/dtonissen/SRMKP/instances.zip.

5. Column generation strategies for solving the LP-relaxation

5.1. Definition of strategies

When we apply column generation to solve the LP-relaxation, we have to maximize the reduced cost of the relevant variables. For the SRD formulation, these are the variables v_{ik} and y_{iq}^s, which indicate whether knapsack filling k is used in the ILP formulation for knapsack i and whether knapsack filling q is used for knapsack i in case of scenario s, respectively. For the CRD formulation, we only need to consider the variables z_{kqi}^s, which indicate

whether the combination of initial knapsack k and recovery knapsack q for scenario s is selected for knapsack i. The formulas for the reduced cost can be found in Sections 3.1 and 3.2, respectively. Moreover, we showed in Section 3.2 that the pricing problem for a given knapsack-scenario combination can be solved using dynamic programming, which in principle yields one column per combination. We can choose, however, how many and which of the pricing problems (subproblems) are solved per iteration and which columns are added to the master problem. Hence, we define different strategies and decide empirically which one is the best.

We define the following basic strategies:

- *Interleaved*: This method goes through the subproblems from knapsack 1 to m and scenario 1 to $|S|$ and solves the subproblems one by one. As soon as we find a subproblem with positive reduced cost, a column is created, added and the master problem is solved; in the next iteration, we continue our search with the next subproblem in the list.
- *Best k*: This method solves the subproblem for all knapsack-scenario combinations, after which the k columns with the highest positive reduced cost are added to the master problem. This method has two special cases: "Best", where only the best column is added and "All", where all columns with positive reduced cost are added to the master problem.
- *SingleK*: This method goes through the knapsacks from 1 to m. In each iteration it solves the subproblems for all scenarios for a specific knapsack. For each subproblem we add the column with highest positive reduced cost to the master problem, after which the master problem gets solved. Then the method continues to the next knapsack.
- *BestK*: This method solves the subproblems for each knapsack-scenario combination and finds the column with the highest reduced cost. It then adds this column to the master problem, together with all other columns with positive reduced cost that were determined for the same knapsack and other scenarios.
- *SingleS*: This method solves the pricing problem for all knapsacks for a specific scenario. All columns for that specific scenario with a positive reduced cost are added to the master problem and the master problem is solved. Then the method continues to the next scenario.
- *BestS*: This method solves all subproblems and finds the subproblem with the highest reduced cost. It then adds the corresponding column to the master problem, together with the columns with positive reduced cost that were determined for the same scenario and other knapsacks.

Next to generating columns by solving the pricing problem, we can generate additional columns to speed up convergence. For the CRD formulation, we can use the fact that the initial solution has to be the same for all scenarios. Suppose that we have solved the pricing problem for scenario s and knapsack i from which we find the feasible knapsack filling k for the initial solution. We can now easily determine the optimal recovery with respect to the initial knapsack filling k for each of the other $|S| - 1$ scenarios in $S \backslash \{s\}$ by solving a knapsack problem, where the item set is restricted to the items available in k. Since we need these columns when we want to use the initial filling k for knapsack i, we add the resulting columns to the master problem. A speed-up based on the same principle was tested by van den Akker et al. (2016).

The SRD formulation does not have this property. However, additional knapsack fillings for the scenarios can be generated from any initial knapsack filling by the same procedure. Unfortunately, given a knapsack filling for a scenario, we are not able to find the best filling for all other scenarios for the same knapsack. But we can find the best initial filling for that knapsack, which has to be a superset of the items from the scenario column. In that case, we

have to find the best filling for the size difference between the initial situation and the scenario; moreover we are only allowed to use items from the complementary set. The initial knapsack filling then becomes the union of the found items and the items in the scenario knapsack filling. With this initial knapsack filling, we can find the knapsack fillings for the remaining scenarios.

5.2. Experiments for the CRD formulation

We implemented our algorithms with the Java programming language and used ILOG CPLEX 12.4 to solve the Linear Programs. An ®Intel ™core i-5-3570K 3.40 GHz processor equipped with 8GB of RAM was used for all experiments.

We started our experiments with the CRD formulation, first without the speed-up and later with the speed-up in which we generate additional columns as described above. All strategies are used for these tests. For the "Best k" method, we used the special cases "Best" and "All", and we further chose k equal to 25%, 50% and 75% of the $m \cdot |S|$ columns that were generated per iteration.

5.2.1. Initial assessment

We used the small test set from Section 4.4, and we allowed a maximum solution time of 1 min per instance; instances that did not solve within that time are registered as a failure. This is necessary because some instances take a long time to solve, while the smallest instances are solved in less than 1 ms. To be able to compare the average solution time of the different methods, we use the maximum allowed solution time as solution time for the instances that we could not solve. Next to the name of the method, we report in our tables: the average number of iterations (*iterations*); the average number of columns that have been added (*columns*); the average number of times we solve the pricing problem (*pricing*); the average time it takes to solve the instances (*time (ms.)*), with a maximum of 1 min per instance; the failure rate (*fail %*).

The results can be found in Table 4. The best methods with respect to the computational time and failure % are "Best 50%" and "Best 75%", respectively. For the instances that are solved by both methods, we see that "Best 50%" performs significantly better than "Best 75%" with respect to the solution time (Wilcoxon signed-rank test done with R version 3.0.1, $p = 0.018$). The method "Best" performs worst, because it spends a lot of time solving the pricing problems, while it is only allowed to add one column per iteration. The added column has the largest reduced cost, and therefore the "Best" method finds the solution with the smallest number of columns of all methods. The "Best 50%" method has the best balance between the number of solved pricing problems and the number of added columns. The methods that are not based on the "Best" method seem to be too restrictive. Adding the best k columns based on their reduced cost seems to be the best. The

Table 4
Average results for the methods for the 1000 instances from the small test set. Note that the averages are only over the instances that are solved for that method.

Method	Iterations	Columns	Pricing	Time (ms.)	Fail %
Interleaved	576.1	601.4	435.6	8230	9.9
Best	261.7	283.5	8628.4	10328	13.9
Best 25%	40.6	936.1	1371.8	5848	6.4
Best 50%	31.7	1256.3	1130.7	5702	5.9
Best 75%	30.0	1267.6	1077.1	5683	6.0
All	29.4	1205.4	1000.6	5780	6.2
SingleS	275.6	742.4	553.3	7229	8.2
SingleK	153.0	1245.8	1024.2	6193	6.2
BestK	66.3	1025.9	2166.7	7468	9.2
BestS	132.6	754.2	4857.1	9127	12.3

Table 5
Results with scenario based speed-up.

Method	Iterations	Columns	Pricing	Time (ms.)	Fail %
InterleavedA	109.8	1787.1	94.9	5159	5.3
BestA	49.1	789.9	1698.5	6002	7.1
BestA 25%	12.2	2210.4	154.7	5091	4.6
BestA 50%	10.3	2222.7	112.1	5981	6.5
BestA 75%	9.8	2313.8	104.5	6227	6.9
AllA	9.4	2302.8	101.0	6458	7.4
SingleSA	53.9	1900.4	99.9	4901	4.9
SingleKA	40.5	2412.7	103.9	6453	7.1
BestKA	21.8	2262.3	269.9	7242	8.0
BestSA	25.2	2110.8	755.6	6028	6.0

"Best 50%" method is faster than the "Best" method by a factor of 4.5 when the solution times of the two methods are compared on the instances that both of them could solve.

5.2.2. Introduction of the speed-up

When the speed-up is included, denoted by an A after the method's name, 6 out of 10 methods perform faster. Furthermore, the methods require fewer iterations and the pricing problem is solved less often, but this comes at the expense of having more columns. The results of the speed-up can be found in Table 5.

The best results are obtained with the "BestA 25%" or the "SingleSA" method. The "SingleSA" method is significantly faster than the "BestA 25%" method for the instances that are solved by both strategies, but "SingleSA" has a larger failure rate. These two strategies are followed by the "InterleavedA" method. The inferior performance of the "SingleKA" method is as expected. The reason is that the pricing problem already considers all scenarios for a specific knapsack, while the speed-up again considers all scenarios for the same knapsack. This means that we can generate a maximum of $|S| \cdot (|S| - 1)$ columns for the same knapsack per iteration. An explanation for the poor performance for the "Best 50%", "Best 75%" and "All" strategies is that they produce many columns of inferior quality (lower reduced costs) per iteration. For each of those columns, $|S| - 1$ additional columns are generated, which are expected to be of inferior quality, too. Consequently, focusing on good (high reduced costs) columns becomes more important, when the speed-up is used.

Comparing the solution time of the strategies is not straightforward due to the unsolved instances. However, we can compare two strategies with each other by taking the average time over the instances that are solved for both strategies. If we compare the "Best 75%" and "BestA 25%" methods in this way then we find an average time of 1870 ms for the "Best 75%" method and 1461 ms for the "BestA 25%" method, which is an improvement by a factor of 1.3. Furthermore, there is also a decrease of 1.3 percentage point in the failure rate. "BestA 25%" performs better than "Best" by a factor of 5.4 for the instances that are solved by both strategies.

We further tweak the "Best k" method by considering more possibilities for k than just multiples of 25% times $m \cdot |S|$. Next to varying the percentage with a step size of multiples of 5%, we test variants of "Best k" in which the number of columns that can be added is a function of either the number of scenarios $|S|$ or the number of knapsacks m. We denote this method by putting this number after the name of the method; for example "BestA $2m$" indicates that we are allowed to add the best $2m$ columns per iteration in method "BestA".

To save space, we do not show the tables. The best result for the methods where we varied the percentage is "BestA 5%" with an average solution time of 4354 ms, where 3.6% of the instances failed. The best result for the methods "Best k", where k is a function of the number of knapsacks is the "Best $2m$" method, with an average solution time of 3621 ms and a failure rate of 2.6%.

Table 6
Results of "BestA k", while varying k.

Method	Time (ms.)	Fail %	Method	Time (ms.)	Fail %
BestA	7816	10.9	BestA m	5196	5.7
BestA 5%	5899	7.7	BestA 1.5m	5035	5.6
BestA 10%	5896	7.7	BestA 2m	5022	5.3
BestA 20%	5598	7.7	BestA 2.5m	4994	5.3
BestA 50%	5998	8.2	BestA 3m	4997	5.3
AllA	6312	8.7	BestA 4m	5035	5.3

Table 7
≥ 50 scenarios results large test set, with the time in seconds.

Method	Iterations	Columns	Pricing	Time (sec.)	Fail %
BestA 0.5m	53.3	15207.1	20676.6	48.4	69.4
BestA m	34.3	18227.3	16577.9	47.3	65.3
BestA 1.5m	27.7	20240.7	15391.5	47.1	64.4
BestA 2m	23.4	22197.2	14924.7	47.8	65.3
BestA 2.5m	21.2	22998.4	14446.3	48.2	66.9
BestA 3m	19.2	24233.5	13767.8	49.4	70.9

Compared to the "Best 25%" method, which performed the best before this additional optimization, this is a significant improvement (Wilcoxon $p = 2.2e^{-16}$) by a factor of 1.5. Furthermore the failure rate decreases by 2%.

Using the number of scenarios to decide how many columns may be added per iteration gives poor results relative to using the number of knapsacks. A possible explanation for this phenomenon is the following. We assume that using the number of knapsacks works well, because of the combination with the speed-up. Remember that for each column, the speed-up generates the columns for the other $|S| - 1$ scenarios, and hence we always add at most $k \cdot m \cdot |S|$ columns. Similarly, when using the number of scenarios as a guide we add a maximum of $k \cdot |S|^2$ columns. Consequently, when using the number of knapsacks we always add a number of columns relative to the minimum number of columns needed for a solution, which is not the case when the scenarios are used.

5.2.3. Influence of the number of scenarios, items, and knapsacks on the methods

We also test the impact of the number of scenarios, knapsacks and items. For this experiment, we used the large varied test set from Section 4, with 8400 instances. We varied k in the method "BestA k" between 0.5 and 4 times the number of knapsacks, and used "BestA", "BestA 5%", "BestA 10%", "BestA 20%," "BestA 50%", and "AllA" for comparison.

The average results for the large test set can be found in Table 6.

We see that the methods that use the number of knapsacks to determine the maximum number of allowed columns per iteration also work better for the larger test set than adding a preset percentage. The "BestA 2.5m" method solves the instances the fastest, but "Best 3m" is very close.

The results where we exclude the instances with fewer than 50 scenarios from the results (354 instances, 21.5 items and 7.3 knapsacks on average), can be found in Table 7.

When we only look at the results with 50 or more scenarios, "BestA 1.5m" has the best performance, instead of "BestA 2.5m" or "BestA 3m". When we take the set with 100 scenarios or more (123 instances) "BestA 0.5m" performs the best.

To test this hypothesis further, we generate 10 instances with an average number of 300 scenarios (range 202–467), where we purposefully keep the number of items and knapsacks small (6 items and 3 knapsacks). These instances may seem small, but these instances have 900 possible pricing problems per iteration on average. The pricing problems can be solved quickly, since only a few

Table 8
Results for additional instances with many scenarios.

	Iterations	Columns	Pricing	Time (ms.)
BestA	25.1	8095	25834	2759
BestA 0.5*m*	13.5	8360	15631	2264
BestA *m*	10.5	12367	15306	4328
BestA 2*m*	9.3	18370	18116	8488

Table 9
Results on larger test set with ≥ 25 items.

Method	Iterations	Columns	Pricing	Time (s)	Fail %
BestA 1.5*m*	29.6	3097.3	2342.3	37.0	53.1
BestA 2*m*	26.7	3563.0	2292.4	36.6	52.3
BestA 2.5*m*	25.5	3772.8	2195.1	36.3	52.1
BestA 3*m*	24.4	4088.9	2184.1	36.1	51.3
BestA 4*m*	23.5	4874.5	2318.0	36.0	50.2

items are available, making these instances excellent for testing the influence of the number of scenarios on the methods. The results can be found in Table 8.

These results indicate that the more scenarios there are for an instance, the more important it becomes to add good columns. Furthermore, we note that even though these instances have 300 scenarios and 900 possible pricing problems per iteration, they are solved quickly.

We did similar experiments to test the influence of the number of items and knapsacks. Since the time needed to solve the pricing problem heavily depends on the number of items, it is preferable to limit the number of times the pricing problem is solved for instances with a large number of items. Consequently, we expect that the more items the instances have, the more columns will be added per iteration. The results for the subset of instances with 25 or more items (639 instances), which have 7.3 knapsacks and 29.8 scenarios on average, can be found in Table 9.

The "BestA 4*m*" method has the smallest computation time for these instances and has the fewest failures, supporting our hypothesis. However, it solves the pricing problem more often on average than the other methods do. We think that this is due to the fact that this method solves many large instances that the other methods could not solve.

We created a test set with many items, namely 100 items, 3 knapsacks and 3 scenarios, to test the influence of the number of items. However, as the computation time grows exponentially with the number of items, only 3 out of the 10 instances could be solved within an hour. Therefore, we decided not to show these results. For non trivial instances it is necessary to increase the number of items, when increasing the number of knapsacks. Consequently, we could not do similar experiments for a large number of knapsacks as the instances became too large to be solved within an hour.

5.3. Experiments for the SRD formulation

Based on the experimental results obtained in the previous subsection for the CRD formulation, we decided not to test all our methods, but just the ones that performed reasonably well for the CRD formulation. Therefore, we only test the methods "Interleaved", "Best", "Best 0.5*m*", "Best *m*", "Best 2*m*", "Best 3*m*" and "All". We apply the speed-up defined in Section 5 in two different ways. In speed-up I we generate additional columns for the initial knapsacks only, whereas in speed-up A we generate additional columns for both the initial knapsacks and for all scenarios.

We first performed some initial experiments to further limit the methods that were tested extensively. For these initial experiments we took 420 instances from the large test set (8400 instances)

from Section 4.4 and we set the maximum solution time at 1 min. The results are found in Table 10.

The "BestI 3*m*" method performed the best and "Interleaved" the worst. When considering only the instances that are solved by both formulations, "BestI 3*m*" runs faster than "Interleaved" by a factor of 11.1. On the basis of the results obtained on these 420 instances, we selected the methods "AllI", "BestI *m*", "BestI 2*m*", "BestI 3*m*", "BestA 0.5*m*" and "BestA *m*" for our extensive computational tests, in which we solve all 8400 instances from the large test set. The results can be found in Table 11.

On the basis of solution time and failure rate we conclude that the best method for the SRD formulation is the "BestI 3*m*" method, followed by the "BestI 2*m*" method. The solution times for the "BestI 3*m*" method are, according to the paired Wilcoxon signed-rank test (with a p-value of 0.01967), significantly better than the solution times for the "BestI 2*m*" method for the instances that are solved by both methods. We compare the SRD and CRD formulations extensively in Section 7.

6. Branch-and-Price optimization

To find an optimal integer solution we apply branch-and-price (Barnhart et al. 1998), which is a special case of branch-and-bound. Our branching strategy is to choose an item and create $m + 1$ nodes, which correspond to the decision of assigning the item to knapsack i ($i = 1, \ldots, m$) or exclude it from all knapsacks. Excluding an item from all knapsacks means that it cannot be in any initial filling and hence not in any recovery filling. If we include the item in a knapsack, then we have to use it initially and it can be in the recovery solution of the knapsack. An upper bound is obtained from the LP-relaxation, which is solved by column generation at each node. It is possible that our solution contains duplicate columns within the nodes of the branch-and-bound tree, because we add additional columns outside the pricing problem. We remove these duplicate columns, just before the child nodes inherit them. We derive an initial lower bound using the following greedy heuristic that is based on the outcome of the LP-relaxation. We sort the items in descending $\frac{revenue}{weight}$ order and the knapsacks in ascending size. We add item $j \in N$ to knapsack $i \in M$ if $x_{ij} > 0.5$ in the solution of the LP-relaxation and when adding the item does not violate the knapsack size. The remaining space in the knapsacks is filled by adding the remaining items greedily to the knapsacks, while keeping the total weight of the items for every knapsack smaller than or equal to the knapsack size.

Again we start with optimizing the CRD formulation. The experiments are done on the first 500 of the 8400 instances of the large test set introduced in Section 4.4. This subset differs from the one with 420 instances used in Section 5.3 to avoid tailoring on a small fixed set. Because we solve the ILP instead of the LP, we increased the time limit from 1 to 5 min. For our initial experiments we employ a depth-first strategy.

Our first experiment is to test whether the order in which we consider the knapsacks in our branching rule matters. The conclusion is that the differences are not significant, except for the option to exclude the item from all knapsacks, which should be considered as their last option. Next, we test the impact of the order in which we consider the items. We test the following options:

- Sort the items in order of ascending/descending weight.
- Sort the items in order of ascending/descending revenue.
- Sort the items in order of ascending/descending $\frac{revenue}{weight}$ ratio.
- Sort the items on the basis of the x values from the LP-relaxation, where the x value for an item i is defined as $\max_j x_{ij}$; the items are then sorted in order of ascending/descending x value, and in order of ascending $|x - 0.5|$ value.

Table 10
The effect of the speed-ups on the results for the column generation SRD formulation.

	Normal		Speed-up I		Speed-up A	
	Time (ms.)	Fail %	Time (ms.)	Fail %	Time (ms.)	Fail %
Interleaved	6271.5	8.1	5173.6	6.4	6706.3	8.6
Best	6357.6	7.9	4267.3	5.2	3437.6	3.3
All	3174.9	3.3	2638.5	2.6	6620.3	8.6
Best $0.5m$	4753.0	6.0	3022.3	3.3	2955.3	2.9
Best m	4164.5	4.8	2804.1	2.9	3004.6	2.9
Best $2m$	4164.5	4.8	2528.1	2.6	3284.4	3.3
Best $3m$	3381.9	3.6	2457.7	2.1	3603.0	3.3

Table 11
Average results for the methods for all 8400 instances for the SRD formulation.

Method	Iterations	Columns	Pricing	Time (ms.)	Fail%
AllI	15.6	1627.8	1264.9	2593.7	2.4
BestI m	28.2	906.3	3852.6	2722.9	2.8
BestI $2m$	20.9	1056.1	2789.0	2534.6	2.4
BestI $3m$	18.8	1176.4	2351.1	2367.0	2.1
BestA $0.5m$	26.6	1473.4	2325.2	3059.8	3.1
BestA m	18.2	1743.1	1487.7	2992.3	3.0

We call the latter ordering the most infeasible branching from now on. The best three methods and the method in which we do not sort (ascending index) can be found in Table 12. In the column 'snode' we report the node in which the optimum solution is found, where we average over all instances that could be solved.

Compared to sorting in ascending index order, we see that the best three sorting methods have fewer iterations, columns, calls to the pricing problem, and nodes, whereas the solution node is found earlier. These are all indications that using one of those three ways to sort is a good idea. Sorting the items in order of descending revenue seems to be the best both in time and in failure rate. This method has more iterations, calls to the pricing problem and nodes than the other methods, which can be explained by the fact that the additional instances that this method solves are more difficult.

Furthermore, we compare the depth-first strategy with the breadth-first and a best bound strategy. For the best bound strategy, we first calculate all the upper bounds of the children, instead of doing this when we expand the child. The node we choose to expand is the node with the highest upper bound. It provides the most space for improvement and, in this way, we hope to find a better lower bound that will prune many unevaluated nodes. The depth-first strategy turns out to perform best. Therefore, our branching strategy is depth-first search, where we sort items in descending revenue, and where excluding the item from all knapsacks is our last option.

For the SRD formulation we have tested the branch-and-price algorithm in a similar way. The conclusion is that depth-first search, in combination with excluding the item from each knapsack as last option, and sorting the items in the most infeasible order is the best strategy.

7. Comparison of the two formulations

7.1. Comparison of the two formulations for the LP-relaxation

We use the large test set from Section 4.4 with 8400 instances to compare the CRD and SRD formulations while solving the LP-relaxation. For both formulations we use the solution method that performed best in the experiments of Section 5. The results are shown in Table 13.

The SRD formulation has an easier pricing problem, but it has to solve it more often than the CRD formulation. Furthermore, the solution value of the LP-relaxation is higher than that of the CRD formulation in 47.4% of the cases, and it finds an integer solution less often. The quality of the LP-relaxation and non-integrality of its solution have no effect on the solution time of the LP-relaxation, but it will have a negative influence on the time required to solve the ILP formulation. The average time to find the solution for the instances that are solved in both formulations is 3.4 times faster for the SRD formulation (544 ms) than for the CRD formulation (1869.6 ms), which is significantly better (Wilcoxon $p = 2.2e^{-16}$). Furthermore, the SRD formulation solves more instances than the CRD formulation, with a failure rate of 2.1% versus 5.3%.

We continue by comparing the formulations with respect to the impact of the number of scenarios, knapsacks and items on the computational performance. To test the effect of the number of scenarios, we group the instances in subsets by defining a range for the number of scenarios; the range is chosen such that each subset contains at least 100 instances, except for the last one, which contains only 18 instances. After solving the instances, we plot the results for each subset in Fig. 1, where one axis corresponds to the average number of scenarios in the subset, whereas the other axis corresponds to the average solution time. Since we consider a lot of different combinations of number of items, knapsacks, and scenarios, Fig. 1 would have been a cloud of data points without the groupings, from which it would have been hard to draw any conclusions. Furthermore, we show in Fig. 1 that the solution times and the failure rates for the SRD formulation are consistently better than for the CRD formulation, irrespective of the number of scenarios.

Next, we consider the influence of the number of knapsacks. In Fig. 2 we show the results. Here we group together the instances with an equal number of knapsacks, where each group contains 1200 instances. In this figure, we indicate the average time and failure rate per group for both methods; each such point corresponds to 1200 instances. Despite the fact that the SRD formulation has m constraints and columns more than the CRD formulation, the SRD formulation performs consistently better, irrespective of the number of knapsacks (Fig. 2).

We also investigate the effect of the number of items. The results are depicted in Fig. 3. Each point corresponds to at least 90 instances with an equal number of items. The results are as expected: the more items, the better the SRD formulation performs compared to the CRD formulation.

The last test is done on an instance set with 10 randomly generated instances with exactly 100 items and 3.0 knapsacks, and 2.1 scenarios on average. The CRD formulation is only able to solve three of the ten instances within one hour. The SRD formulation can solve all these instances and is faster by a factor 94.6 for the instances they both solved. The SRD formulation is even able to solve 4 out of 5 instances with up to 200 items (3 knapsacks, 3 scenarios) within an hour.
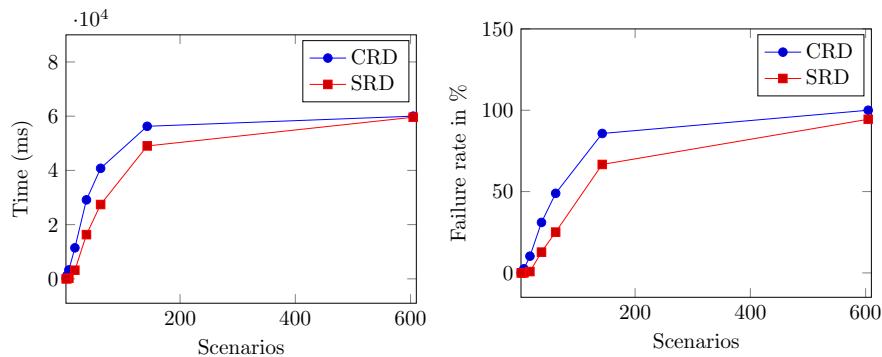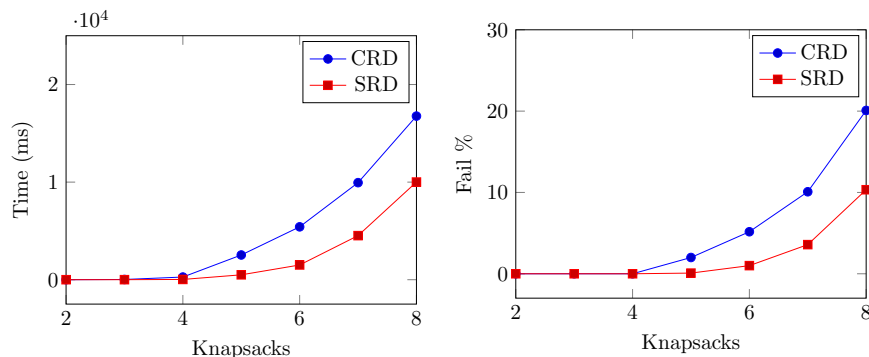
Table 12
The best three results of item sorting compared to sorting them in ascending index.

Sort	Iterations	Columns	Pricing	Nodes	snode	Time (sec.)	Fail%
Ascending index	4171.8	291255.8	71594.8	841.9	621.4	93.4	10.0
Most infeasible	1949.0	188481.2	47518.6	329.7	172.3	80.4	8.2
Descending weight	1679.0	233156.7	33962.1	299.9	200.6	76.4	7.8
Descending revenue	2297.8	215708.6	48719.7	451.0	180.9	74.1	7.0

Table 13
Results for solving the LP-relaxation for the large test set with the CRD and SRD formulations.

Method	Iterations	Columns	Pricing	Integer %	Time (ms)	Fail %
CRD: BestA $2.5m$	13.2	1767.1	1304.5	55.7	4994	5.3
SRD: BestI $3m$	18.8	1176.4	2351.1	39.4	2367	2.1

Fig. 1. Solution time and failure rate in % for the CRD and SRD formulations per scenario.

Fig. 2. Solution time and failure rate in % for the CRD and SRD formulations per knapsack.

Table 14
Results for the CRD and SRD formulations for the structured test set.

Method	LP time (s)	Integer %	Nodes	ILP time (sec.)	Fail %
CRD BestA $2.5m$	31.1	61.6	675.3	103.8	34.5
SRD BestI $3m$	12.8	38.0	1821.3	123.6	41.1

We can conclude that the SRD formulation performs better than the CRD formulation for solving the LP-relaxation.

7.2. Comparison of the two formulations for finding the integer solution

Finally, we test the performance of the SRD and CRD formulations when using branch-and-price to solve the integer problem. For testing, we use the structured test set mentioned in Section 4.4, with a maximum solution time of 300 s. The results of this test can be found in Table 14.

As before, the SRD formulation outperforms the CRD formulation for the LP-relaxation: the SRD formulation solves the in-stances 2.4 times faster than the CRD formulation. However, the LP-relaxation of the CRD formulation is integral for 61.6% of the instances, whereas this holds for only 38.0% of the instances for the SRD formulation. The number of nodes for the CRD formulation is smaller by a factor of 2.7, solving the ILP is faster by a factor of 1.2, and the ILP time is a factor 1.2 faster when compared to the SRD formulation. In addition, the CRD formulation has fewer columns, fewer iterations, and a smaller failure rate of 28.5% (814 instances) versus 34.8% (995 instances). However, the difference in solution time between the instances that are solved by both formulations (62.5%) is not significant according to the Wilcoxon signed-rank test ($p = 0.72$).
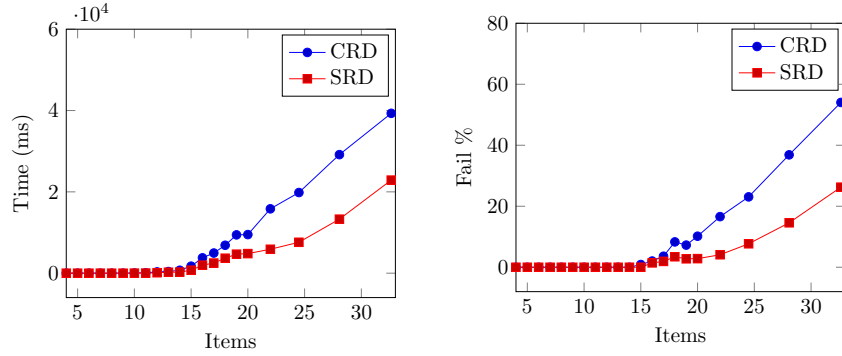
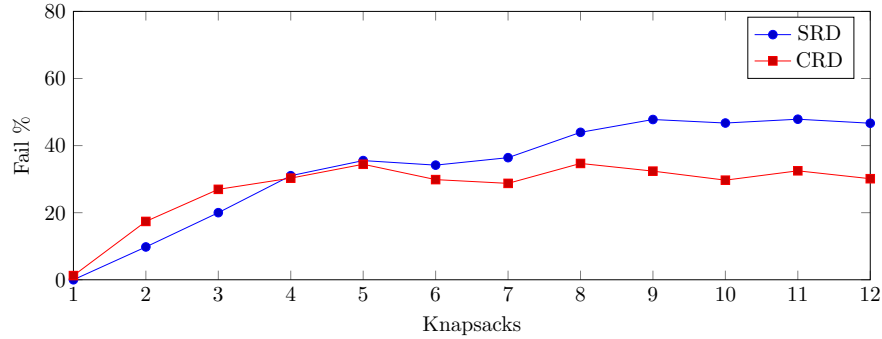**Fig. 3.** Solution time and failure rate in % for the CRD and SRD formulations per item.

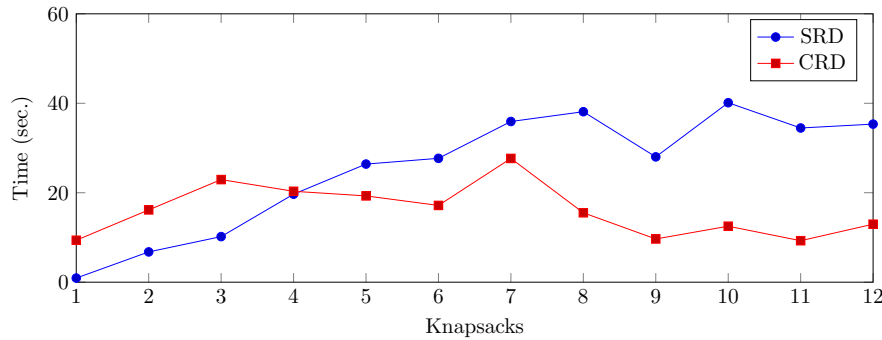**Fig. 4.** Failure rate in % per number of knapsacks per method.

Fig. 5. Time in seconds per number of knapsacks for the SRD and CRD formulations.

Next, we compare the different formulations on instance properties such as the number of scenarios, knapsacks, and items. We notice that the CRD formulation has fewer failures, uses fewer nodes on average, and finds the solution faster, irrespective of the number of scenarios and items.

With respect to the knapsacks, the CRD formulation also generates fewer nodes on average. However, it should be noted that the SRD formulation performs better than the CRD formulation on the instances with up to four knapsacks, while the CRD formulation performs better with more than four knapsacks. Fig. 4 depicts the failure rates for both formulations, while Fig. 5 depicts the solution time in seconds for the instances that both formulations were able to solve.

For the instances with only one knapsack, the SRD formulation is 10.2 times faster, while for the instances with 11 knapsacks the CRD formulation is 3.7 times faster. We can conclude that the more knapsacks the better the CRD formulation performs compared to the SRD formulation. In Table 15 we compare the percentage of instances for which the LP-relaxation is solved with an integer solution, the average and maximum integrality gap, and the number of nodes for the CRD and SRD formulations for every number of knapsacks. We only consider instances that both formulations solved, and the ILP gap is defined as $\frac{Z_{LP}}{Z_{ILP}}$, i.e. the ratio of the LP and ILP value.

For the instances that are solved within 300 s, both decomposition formulations have very small integrality gaps: the maximum gaps are 1.086 and 1.139, while the average gaps are 1.004 and 1.008 for the CRD and SRD formulation, respectively. The CRD formulation performs consistently better than the SRD formulation. Irrespective of the number of knapsacks the CRD formulation finds more integral solutions for the LP-relaxation for the structured test set, the integrality gap is smaller, and it needs fewer nodes. Consequently, the integrality gap cannot cause the CRD formulation to outperform the SRD formulation when the number of knapsacks increases, and therefore we conclude that there must be another reason for this phenomenon.

The most logical explanation for it is the complexity of the pricing problem; recall that the SRD pricing problem is solved in $\mathcal{O}(\min(n \cdot b_i, 2^n))$ time, whereas that of the CRD formulation requires $\mathcal{O}(\min(n \cdot b_i, 2^n))$ for the knapsacks with unaltered size,

Table 15

The effect of the number of knapsacks on the integer % and ILP gap.

Knapsacks	CRD formulation				SRD formulation			
	Integer %	ILP gap	Max gap	Nodes	Integer %	ILP gap	Max gap	Nodes
all	61.4	1.004	1.086	696.9	38.1	1.008	1.139	1235.6
1	99.4	1.000	1.006	0.1	62.3	1.001	1.017	15.5
2	58.1	1.005	1.047	534.6	32.1	1.006	1.047	358.3
3	50.0	1.007	1.086	576.9	40.3	1.008	1.094	247.8
4	45.3	1.007	1.068	846.2	32.1	1.011	1.139	1298.8
5	54.8	1.004	1.029	615.7	38.4	1.008	1.072	939.4
6	57.6	1.006	1.079	862.5	37.1	1.010	1.108	1190.1
7	61.6	1.004	1.037	1727.9	37.1	1.007	1.074	4094.9
8	65.9	1.004	1.047	889.9	29.6	1.010	1.079	1866.6
9	59.8	1.004	1.061	764.3	34.6	1.009	1.103	2566.5
10	62.2	1.004	1.054	638.7	25.8	1.010	1.060	445.0
11	62.9	1.003	1.038	471.4	28.3	1.008	1.064	1368.1
12	63.5	1.004	1.040	232.0	30.2	1.009	1.093	371.6

Table 16

Results larger instances CRD and SRD formulations, with the time expressed in minutes.

Method	LP time (min.)	Nodes	ILP time (min.)	Fail %
CRD BestA $2.5m$	4.0	40081	43.8	63.0
SRD BestI $3m$	1.1	37803	47.0	73.2

and $\mathcal{O}(\min(n \cdot b_i^s \cdot b_i, 3^n))$ for the knapsacks that decrease in size. Monte Carlo simulations show that for a small number of knapsacks the fraction of hard pricing problems ($\mathcal{O}(\min(n \cdot b_i^s \cdot b_i, 3^n))$) is larger. When $m = 1$ there is only a difficult pricing problem, when $m = 2$ we have that 77% of the pricing problems are hard, when $m = 3, 4, 5$ this is approximately 69%, 66%, 65% respectively. When m increases further the number of hard pricing problems is approximately 64%. Furthermore, the average knapsack size becomes smaller when m increases to avoid trivial problems.

7.3. Results for larger instances

To gain more insight in how the formulations solve more difficult instances, we consider the instances with 25 items from the structured test set and we allow a maximum solution time of 60 min. This subset consists of 127 instances, with 7 knapsacks and 41.9 scenarios on average. Both formulations are able to solve the LP-relaxation for all instances. The results can be found in Table 16.

The SRD method solves the LP-relaxation faster than the SRD formulation, but the CRD formulation solves the ILP faster and more instances get solved. Because there are only 26 instances that are solved by both formulations, it is hard to make any conclusions about the number of nodes, iterations and columns of both formulations. For example the LP-relaxation of the CRD method is stronger, but it still uses more nodes on average for the instances it solves. The reason is that the CRD formulation is able to solve larger (and more) instances than the SRD formulation. These larger instances require more nodes to solve, which increases the average. For the instances that are solved by both formulations, the SRD formulation surprisingly performs faster (43.2 min vs 51.4 min). However, based on the failure rate the CRD formulation performs better.

The reason why so few instances (20.5%) are solved by both methods, is that each method is best suited for different types of instances. Again we make a distinction based on the number of knapsacks. For the instances with only one knapsack, the SRD formulation performs more than 300 times faster (0.02 versus 6.3 min), while for 5 knapsacks and more the CRD formula-

tion performs a bit faster and has fewer failures. These results are shown in Fig. 6.

8. Final remarks

In this paper, we introduce several column generation strategies and different clever ways to generate additional columns outside of the pricing problem. Optimizing the combination of these strategies appears to be very important for the SMKP. It decreases the solution time by a factor of 10.5, and the failure percentage by 11.3 points for the CRD formulation (small test set). For the SRD formulation, it decreases the solution time by factor of 11.1 and the failure percentage by 6.0 points (large test set). Based on our earlier paper (van den Akker et al. 2016), we expected this kind of results for the CRD formulation, but we show that it improves the SRD formulation in a comparable manner too. Producing additional columns for each generated column decreases the solution time for the SRD formulation, but we obtain better results when we only generate additional columns for the initial columns. We expect that the technique of creating additional columns will work for most or even all problems that have a discrete set of scenarios, but this may depend on the recovery algorithm.

In terms of solution time for the LP-relaxation, the SRD formulation outperformed the CRD formulation by a factor of 3.4, and a decrease in failure percentage of 3.2 points for the structured test set. On the other hand, however, the optimum solution of the LP-relaxation of the CRD formulation was integral for 55.7% of the instances, whereas for the SRD formulation this was the case for only 39.4% of the instances. For the integer linear program, more instances were solved with the CRD formulation than with the SRD formulation (71.5% versus 65.2%). However, there is no significant difference between the solution times for the instances that were solved by both. For the number of knapsacks, there is a performance difference between the CRD and the SRD formulation. For four or fewer knapsacks, the SRD formulation performs better, while for five or more knapsacks, the CRD formulation is superior. For the instances with only one knapsack, the SRD formulation is 10.2 times faster, while for the instances with 11 knapsacks, the CRD formulation is 3.7 times faster. No such differences can be found for the number of scenarios and items; in all cases the CRD formulation appears to perform better. This is in line with our earlier result that the performance is knapsack related.

The CRD formulation has a stronger LP-relaxation and more integer solutions. However, the number of knapsacks appears to be the main factor that determines which formulation is computationally better for the SMKP. The most logical explanation for it is the complexity of the pricing problem. For the SRD formulation the pricing problem is solved in $\mathcal{O}(\min(n \cdot b_i, 2^n))$ time, whereas that
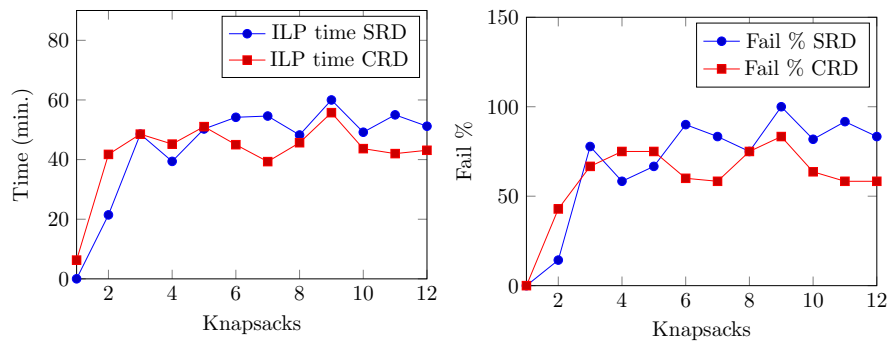
Fig. 6. Solution time in minutes and failure rate in % for the CRD and SRD formulations per knapsack.

of the CRD formulation requires $\mathcal{O}(\min(n \cdot b_i, 2^n))$ for the knapsacks with unaltered size, and $\mathcal{O}(\min(n \cdot b_i^s \cdot b_i, 3^n))$ for the knapsacks that decrease in size. Monte Carlo simulations show that for a small number of knapsacks the fraction of hard pricing problems $(\mathcal{O}(\min(n \cdot b_i^s \cdot b_i, 3^n)))$ is larger. When $m = 1$ there is only a difficult pricing problem, when $m = 2$ we have that 77% of the pricing problems are hard, and this converges to approximately 64% when m increases. Furthermore, the average knapsack size becomes smaller when m increases to avoid trivial problems.

We expect that larger instances can be solved by employing techniques to prune more nodes in the branch-and-bound tree and by using parallelization. The nodes in the branching tree often contain symmetry, which is especially the case when there are knapsacks with the same size or items with the same weight and size. Methods that exploit this symmetry can be used to prune more nodes. It may also be possible to exploit the dominance relations between different items. Furthermore, there is considerable overlap in the pricing problems, especially when the scenarios are similar. Finding a way to exploit this overlap is expected to decrease the solution time considerably. We did not implement parallelization because we focused on the properties of the formulation. However, all columns at an iteration are independent of each other and therefore can be calculated at the same time.

Another option to investigate is heuristic methods. If a greedy method is used such as removing the items in order of their revenue or revenue to weight ratio, then we can calculate the initial and all scenario knapsack fillings within a single pricing problem. This method is described in Section 5.3.4. of the Masters' thesis of Bouman (2011) for the single two-stage stochastic knapsack. This method can easily be adapted to multiple knapsacks and we expect that it will solve our instances faster.

Acknowledgments

References

Akker, J.M., Bouman, P., Hoogeveen, J., Tönissen, D., 2016. Decomposition approaches for recoverable robust optimization problems. Eur. J. Oper. Res. 251 (3), 739–750.

Barnhart, C., Johnson, E., Nemhauser, G., Savelsbergh, M., Vance, P., 1998. Branch-and-price: column generation for solving huge integer programs. Oper. Res. 46, 316–329.

Bhalgat, A., Goel, A., Khanna, S., 2011. Improved approximation results for stochastic knapsack problems. In: Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms. SIAM, pp. 1647–1665.

Bouman, P., 2011. A Column Generation Framework for Recoverable Robustness. Department of Information and Computing Sciences, Utrecht University Master's thesis.

Büsing, C., Koster, A., Kutschka, M., 2011a. Recoverable Robust Knapsacks: Γ-Scenarios. In: Pahl, J., Reiners, T., Voß, S. (Eds.). Lecture Notes in Computer Science, Network Optimization, 6701 doi:10.1007/978-3-642-21527-8_65.

Büsing, C., Koster, A., Kutschka, M., 2011b. Recoverable robust knapsacks: the discrete scenario case. Optim. Lett. 5, 379–392.

Cacchiani, V., Caprara, A., Galli, L, Kroon, L., Maróti, G., 2008. Recoverable robustness for railway rolling stock planning. OASIcs-OpenAccess Series in Informatics, 9. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Carraway, R., Schmidt, R., Weatherford, L., 1993. An algorithm for maximizing target achievement in the stochastic knapsack problem with normal returns. Naval Res. Logist. (NRL) 40 (2), 161–173.

Chen, K., Ross, S., 2014. An adaptive stochastic knapsack problem. Eur. J. Oper. Res. 239 (3), 625–635.

Cicerone, S., DAngelo, G., Di Stefano, G., Frigioni, D., Navarra, A., Schachtebeck, M., Schöbel, A., 2009. Recoverable robustness in shunting and timetabling. In: Robust and Online Large-Scale Optimization. Springer, pp. 28–60.

Dean, B.C., Goemans, M.X., Vondrák, J., 2008. Approximating the stochastic knapsack problem: the benefit of adaptivity. Math. Oper. Res. 33 (4), 945–964.

Fukunaga, A., 2011. A branch-and-bound algorithm for hard multiple knapsack problems. Ann. Oper. Res. 184, 97–119.

Gaivoronski, A., Lisser, A., Lopez, R., Xu, H., 2011. Knapsack problem with probability constraints. J. Global Optim. 49 (3), 397–413.

Goel, A., Indyk, P., 1999. Stochastic load balancing and related problems. In: Foundations of Computer Science, 1999. 40th Annual Symposium on. IEEE, pp. 579–586.

Henig, M., 1990. Risk Criteria in a Stochastic Knapsack Problem. Oper. Res. 38 (5), 820–825.

Kellerer, H., Pferschy, U., Pisinger, D., 2004. Knapsack Problems. Springer.

Kleinberg, J., Rabani, Y., Tardos, É., 2000. Allocating bandwidth for bursty connections. SIAM J. Comput. 30 (1), 191–217.

Kleywegt, A., Papastavrou, J., 2001. The dynamic and stochastic knapsack problem with random sized items. Oper. Res. 49 (1), 26–41.

Kosuch, S., 2014. Approximability of the two-stage stochastic knapsack problem with discretely distributed weights. Discrete Appl. Math. 165, 192–204.

Kosuch, S., Lisser, A., 2010. Upper bounds for the 0–1 stochastic knapsack problem and a b&b algorithm. Ann. Oper. Res. 176 (1), 77–93.

Kosuch, S., Lisser, A., 2011. On two-stage stochastic knapsack problems. Discrete Appl. Math. 159 (16), 1827–1841.

Li'ang, Z., Suyun, G., 1986. Complexity of the 0/1 multiple knapsack problem. J. Comput. Sci. Technol. 1, 46–50.

Liebchen, C., Lübbecke, M., Möhring, R., Stiller, S., 2009. The concept of recoverable robustness, linear programming recovery, and railway applications. In: Robust and Online Large-Scale Optimization. In: Lecture Notes in Computer Science, 5686. Springer Berlin Heidelberg, pp. 1–27.

Martello, S., Toth, P., 1980. Solution of the zero-one multiple knapsack problem. Eur. J. Oper. Res. 4 (4), 276–283.

Martello, S., Toth, P., 1981. A bound and bound algorithm for the zero-one multiple knapsack problem. Discrete Appl. Math. 3, 275–288.

Morton, D., Wood, R., 1998. On a stochastic knapsack problem and generalizations. In: Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search. Springer, pp. 149–168.

Pisinger, D., 1999. An exact algorithm for large multiple knapsack problems. Eur. J. Oper. Res. 114, 528–541.

Pisinger, D., 2004. Where are the hard knapsack problems? Comput. Oper. Res. 32, 2272–2284.