Recoverable robustness by column generation

P.C. Bouman

J.M. van den Akker

J.A. Hoogeveen

Technical Report UU-CS-2011-013
May 2011

Department of Information and Computing Sciences
Utrecht University, Utrecht, The Netherlands
www.cs.uu.nl

ISSN: 0924-3275

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80.089
3508 TB Utrecht
The Netherlands

Recoverable robustness by column generation

P.C. Bouman, J.M. van den Akker, J.A. Hoogeveen

Department for Information and Computing Sciences
Utrecht University
Princetonplein 5, 3584 CC Utrecht, The Netherlands
pcbouman@cs.uu.nl, J.M.vandenAkker@cs.uu.nl, J.A.Hoogeveen@cs.uu.nl

Abstract. Real-life planning problems are often complicated by the occurrence of disturbances, which imply that the original plan cannot be followed anymore and some recovery action must be taken to cope with the disturbance. In such a situation it is worthwhile to arm yourself against common disturbances. Well-known approaches to create plans that take possible, common disturbances into account are robust optimization and stochastic programming. Recently, a new approach has been developed that combines the best of these two: recoverable robustness. In this paper, we apply the technique of column generation to find solutions to recoverable robustness problems. We consider two types of solution approaches: separate recovery and combined recovery. We show our approach on two example problems: the size robust knapsack problem, in which the knapsack size may get reduced, and the demand robust shortest path problem, in which the sink is uncertain and the cost of edges may increase.

1 Introduction

Most optimization algorithms rely on the assumption that all input data are deterministic and known in advance. However, in many practical optimization problems, such as planning in public transportation or health care, data may be subject to changes. To deal with this uncertainty, different approaches have been developed. In case of *robust optimization* [2] we choose the solution with minimum cost that remains feasible for a given set of disturbances in the parameters. In case of *stochastic programming* [3], we take *first stage decisions* on basis of the current information and, after the value of the unknown data has been revealed, we take the *second stage* or *recourse decisions*. The objective here is to minimize the cost of the first stage decisions plus the expected cost of the recourse decisions. The recourse decision variables may be restricted to a polyhedron through the so-called technology matrix [3]. So robust optimization wants the initial solution to be completely immune for a predefined set of disturbances, while stochastic programming includes a lot of options to postpone decisions to a later stage or change decisions in a later stage.

Recently, the notion of *recoverable robustness* [11] has been developed, which combines robust optimization and second-stage recovery options. Recoverable

robust optimization computes solutions, which for a given set of scenarios can be recovered to a feasible solution according to a pre-described, fast, and simple recovery algorithm. The main difference between recoverable robustness and stochastic programming is the way in which recourse actions are limited. The property of recoverable robustness that recourse actions must be achieved by applying a simple algorithm instead of being bounded by a polyhedron makes it very suitable for combinatorial problems. As an example, consider the planning of buses and drivers in a large city. We may expect that during rush hour buses may be delayed, and hence may be too late to perform the next trip in their schedule. In case of robust optimization, we can counter this only by making the time between two consecutive trips larger than the maximum delay that we want to take into account. In case of recoverable robustness, we are allowed to change, if necessary, the bus schedule, but this is limited by the choice of the recovery algorithm. For example, we may schedule a given number of stand-by drivers and buses, which can take over the trip of a delayed driver/bus combination. Especially in the area of railway optimization (see e.g. [6] and [7]) recoverable robust optimization methods have gained a lot of attention.

In this paper we present a new approach for solving recoverable robust optimization problems. We use *column generation* for recoverable robust optimization. We will present column generation models for the size robust knapsack problem and for the demand robust shortest path problem. Our approach can be generalized to many other problems. To the best of our knowledge, this is the first paper applying column generation to recoverable robust optimization. Another decomposition approach, namely Benders decomposition, is used in [5] to assess the Price of Recoverability for recoverable robust rolling stock planning in railways.

The remainder of the paper is organized as follows. In Section 2, we define the concept of recoverable robustness. In Section 3, we consider the size robust knapsack problem and in Section 4 the demand robust shortest path problem. In Section 5, we report on computational results. Finally, Section 6 concludes the paper.

2 Recoverable robustness

In this section we formally define the concept of recoverable robustness. We are given an optimization problem

$$P = \min\{f(x)|x \in F\},$$

where $x \in \mathbb{R}^n$ are the decision variables, f is the objective function and F is the set of feasible solutions.

Disturbances are modeled by a set of scenarios S. We use F_s to denote the set of feasible solutions for scenario $s \in S$, and we denote the decision variables for scenarios s by y^s. The set of algorithms that can be used for recovery are denoted by $\mathcal{A}$, where $A(x, s) \in \mathcal{A}$ determines a feasible solution y^s from a given initial solution x in case of scenario s. In case of planning buses and drivers a

scenario corresponds to a set of bus trips that are delayed, and the algorithms in $\mathcal{A}$ decide about the use of standby drivers.

The *recovery robust* optimization problem is now defined as:

$$\mathcal{RRP}_{\mathcal{A}} = \min\{f(x) + g(\{y^s | s \in S\}) | x \in F,\ A \in \mathcal{A},\ \forall_{s \in S} y^s = A(x, s)\}.$$

Here, $g(\{y^s | s \in S\})$ denotes the cost associated with the recovery variables y^s. There are many possible choices for g. A few examples are as follows:

1. g is defined as the all-zero function. This models the situation where our only concern is the feasibility of the recovered solutions.
2. g is equal to the maximal cost of the recovered solutions y^s. This corresponds to minimizing the worst-case cost.
3. g measures the deviation of the solutions y^s from the initial solution x. Note that this deviation may also be limited by the recovery algorithms.
4. Suppose we are given the probabilities p_s of scenarios s. Then g is defined as expected value of the solution after recovery, i.e., $g(\{y^s | s \in S\}) = \sum_{s \in S} p_s g(y^s)$, where $g(y^s)$ is the cost of solution y^s.

Although earlier papers on recoverable robustness (e.g. [11]) consider the latter type of definition of g as two-stage stochastic programming, we think that the requirement of a pre-described easy recovery algorithms makes this definition fit into the framework of recoverable robustness.

3 Size robust knapsack problem

We consider the following knapsack problem. We are given n items, where item j ($j = 1, \ldots, n$) has revenue c_j and weight a_j. Each item can be selected at most once. The knapsack size is b. We define the *size robust knapsack problem* as the knapsack problem where the knapsack size b is subject to uncertainty. We denote by $b_s < b$ the size of the knapsack in scenario $s \in S$. We assume that the knapsack will keep its original size with probability p_0 and that scenario s will occur with probability p_s. We study the situation in which recovery has to be performed by removing items. As soon as it becomes clear which scenario appears, we can find for a given initial solution the optimal recovery by dynamic programming. Recently [4] have studied an extended version of our knapsack problem. They show $\mathcal{NP}$-hardness of several variants and develop a polyhedral approach to solve these problems.

We are going to discuss two decomposition approaches for the size robust knapsack problem. In both cases we reformulate the problem such that we have to select one knapsack filling for the initial problem and all scenarios from a given set. The difference consists of the way we deal with the scenarios. In *Separate Recovery Decomposition*, we select an initial knapsack filling and separately we select a knapsack filling for each scenario; the relation that the initial knapsack filling should contain all scenario fillings is enforced by constraints in the master problem. In *Combined Recovery Decomposition*, we select for each scenario a

combination of an initial knapsack filling together with the optimal recovery knapsack for that single scenario. We enforce that only one initial knapsack filling will get selected in the master problem.

3.1 Separate Recovery Decomposition

We define $K(b)$ as the set of feasible knapsack fillings with size at most b. For $k \in K(b)$, we denote its revenue by $C_k = \sum_{i \in k} c_i$. In the same way, we denote the revenue of $k \in K(b_s)$ by $C_k^s = \sum_{i \in k} c_i$.

We define decision variables

$$x_k = \begin{cases} 1 \text{ if knapsack } k \in K(b) \text{ is selected,} \\ 0 \text{ otherwise.} \end{cases}$$

and

$$y_k^s = \begin{cases} 1 \text{ if knapsack } k \in K(b_s) \text{ is selected for scenario } s, \\ 0 \text{ otherwise.} \end{cases}$$

The problem can now be formulated as follows. This is called the Master ILP.

$$\max p_0 \sum_{k \in K(b)} C_k x_k + \sum_{s \in S} p_s \sum_{k \in K(b_s)} C_k^s y_k^s$$

subject to

$$\sum_{k \in K(b)} x_k = 1 \tag{1}$$

$$\sum_{k \in K(b_s)} y_k^s = 1 \tag{2}$$

$$\sum_{k \in K(b)} a_{ik} x_k - \sum_{k \in K(b_s)} a_{ik}^s y_k^s \geq 0 \quad \text{for all } i \in \{1, 2, \ldots, n\},\ s \in S \tag{3}$$

$$x_k, \in \{0, 1\} \quad \text{for all } k \in K(b) \tag{4}$$

$$y_k^s, \in \{0, 1\} \quad \text{for all } k \in K(b_s),\ s \in S, \tag{5}$$

where the index variables a_{ik} and a_{ik}^s are defined as follows:

$$a_{ik} = \begin{cases} 1 \text{ if item } i \text{ is in knapsack } k \in K(b), \\ 0 \text{ otherwise.} \end{cases}$$

and

$$a_{ik}^s = \begin{cases} 1 \text{ if item } i \text{ is in knapsack } k \in K(b_s), \\ 0 \text{ otherwise.} \end{cases}$$

In the above model constraint (1) states that exactly one knapsack is selected for the original situation and constraints (2) that exactly one knapsack is selected for each scenario. Constraints (3) ensures that recovery is done by removing items.

We want to solve this ILP formulation using Branch-and-Price [1]. We relax the integrality constraints (4) and (5) into $x_k \geq 0$ and $y_k^s \geq 0$, and solve this LP-relaxation. To deal with the large number of variables we are going to solve the problem by *column generation*. We start with a limited subset of the variables and solve the LP-relaxation for this subset only; this is called the restricted master LP. Then we solve the pricing problem, i.e., we look for variables that are not yet included in the restricted master LP and can to improve the solution if their value is made positive. If such variables are found, they are added to the restricted master LP, it is solved again, after which pricing is performed etc. If pricing does not find any new variables we know that the master LP has been solved to optimality.

The pricing problem
From the theory of linear programming it is well-known that for a maximization problem increasing the value of a variable will improve the current solution if and only if its reduced cost is positive. The pricing problem then boils down to maximizing the reduced cost.

Let λ, μ_s, and π_{is} be the dual variables of constraints 1, 2, and 3 respectively. Now the reduced cost $c^{\text{red}}(x_k)$ of x_k is given by

$$\begin{aligned} c^{\text{red}}(x_k) &= p_0 \sum_{i \in k} c_i - \lambda - \sum_{i=1}^{n} \sum_{s \in S} a_{ik} \pi_{is} \\ &= \sum_{i=1}^{n} a_{ik} (p_0 c_i - \sum_{s \in S} \pi_{is}) - \lambda. \end{aligned}$$

The pricing problem is to find a feasible knapsack for the original scenario, where the revenue of item i, equals $(p_0 c_i - \sum_{s \in S} \pi_{is})$. This is just the original knapsack problem with modified objective coefficients. Similarly the reduced cost $c^{\text{red}}(y_k^s)$ are given by:

$$c^{\text{red}}(y_k^s) = \sum_{i=1}^{n} a_{ik}^s (p_s c_i + \pi_{is}) - \mu_s.$$

It follows that the pricing is exactly the knapsack problem with knapsack size b_s and modified objective coefficients. Note that in the pricing problem an item may have a negative revenue. Clearly such items can be discarded.

To find an integral solution, we are going to apply Branch-and-Price. We branch on items that are fractional in the current solution; this is easily combined with column generation.

3.2 Combined Recovery Decomposition

In contrast to the *Separate Recovery Decomposition*, we consider fillings of the initial knapsack in combination with the optimal recovery for *one* scenario. Consequently, we introduce decision variables:

$$z_{kq}^s = \begin{cases} 1 \text{ if the combination of initial solution } k \in K(b) \\ \quad \text{and recovery solution } q \in K(b_s) \text{ is selected for scenario } s, \\ 0 \text{ otherwise.} \end{cases}$$

The ILP model further contains the original variable x_i signaling if item i is contained in the initial knapsack. We can formulate the problem as follows:

$$\max p_0 \sum_{i=1}^{n} c_i x_i + \sum_{s \in S} p_s \sum_{(k,q) \in K(b) \times K(b_s)} C_q^s z_{kq}^s$$

subject to

$$\sum_{(k,q) \in K(b) \times K(b_s)} z_{kq}^s = 1 \quad \text{for all } s \in S \tag{6}$$

$$x_i - \sum_{(k,q) \in K(b) \times K(b_s)} a_{ik} z_{kq}^s = 0 \quad \text{for all } i \in \{1, 2, \ldots, n\},\ s \in S \tag{7}$$

$$x_i, \in \{0, 1\} \quad \text{for all } i \in \{1, 2, \ldots, n\} \tag{8}$$

$$z_{kq}^s, \in \{0, 1\} \quad \text{for all } k \in K(b),\ q \in K(b_s),\ s \in S, \tag{9}$$

Constraints (6) enforce that exactly one combination is selected for each scenario; constraints (7) ensure that the same initial knapsack filling is selected for all scenarios.

Again, we are going to solve the LP-relaxation by column generation. We include the variables x_i in the restricted master LP and, hence pricing is only performed for the variables z_{kq}^s. We denote the dual variables of constraints (6) and (7) by ρ_s and σ_{is}, respectively. The reduced cost of z_{kq}^s is now equal to:

$$c^{\text{red}}(z_{kq}^s) = \sum_{i=1}^{n} a_{iq}^s p_s c_i + \sum_{i=1}^{n} a_{ik} \sigma_{is} - \rho_s.$$

We solve the pricing problem for each scenario separately. We have to find an initial and recovery solution. This can be solved by dynamic programming. The main observation is that there are three types of items: items included in both the initial and recovery knapsack, items selected for the initial knapsack, but removed by the recovery, and non-selected items. We define state variables $D(i, w_0, w_s)$ as the best value for a combination of an initial and recovery knapsack for scenario s, such that the initial knapsack is a subset of $\{1, 2 \ldots, i\}$, the recovery knapsack is a subset of the initial knapsack, and the initial and recovery knapsack have weight w_0 and w_s, respectively. The recurrence relation is as follows:

$$D(i, 0, 0) = 0 \quad \forall i$$

$$D(0, w_0, w_s) = -\infty \quad \text{for } w_0, w_s > 0$$

$$D(i, w_0, w_s) = \max \begin{cases} D(i-1, w_0, w_s) \\ D(i-1, w_0 - a_i, w_s) + \sigma_{is} \\ D(i-1, w_0 - a_i, w_s - a_i) + \sigma_{is} + p_s c_i \end{cases}$$

4 Demand robust shortest path problem

The demand robust shortest path problem is an extension of the shortest path problem and has been introduced in [8]. We are given a graph (V, E) with cost c_e on the edges $e \in E$, and a source node $v_{\text{source}} \in V$. The question is to find the cheapest path from source to the sink, but the exact location of the sink is subject to uncertainty. Moreover, the cost of the edges may change over time. More formally, there are multiple scenarios $s \in S$ that each define a sink v^s_{sink} and a factor $f^s > 1$ by which the cost of the edges are scaled.

[12] has studied two variants of this problem. In both cases, the sink is known, but the costs of the edges can vary. Initially, a path has to be chosen. In the first variant, recovery is limited by replacing at most k edges in the chosen path; [12] shows this problem to be $\mathcal{NP}$-hard. In the second variant any new path can be chosen, but you get a discount on already chosen edges; [12] looks at the worst case behavior of a heuristic.

In contrast to [12], we can buy any set of edges in the initial planning phase. In the recovery phase, we have to extend the initial set such that it contains a path from the source to the sink v^s_{sink}, while paying increased cost for the additional edges. Our objective is to minimize the cost of the worst case scenario. Remark that, when the sink gets revealed, the recovery problem can be solved as a shortest path problem, where the edges already bought get zero cost.

Observe that the recovery problem has the constraint that the union of the edges selected during recovery and the initially selected edges contains a path from source v_{source} to sink v^s_{sink}. This constraint is hard to express in Separate Recovery Decomposition, but it fits very well in Combined Recovery Decomposition.

Our Combined Recovery Decomposition model contains the variable x_e signaling if edge $e \in E$ is selected initially. Moreover, for each scenario, it contains variables indicating which edges are selected initially and which edges are selected during the recovery:

$$z^s_{kq} = \begin{cases} 1 \text{ if the combination of initial edge set } k \subseteq E \\ \quad \text{and recovery edge set } q \subseteq E \text{ is selected for scenario } s, \\ 0 \text{ otherwise.} \end{cases}$$

Observe that z^s_{kq} is only defined if k and q are feasible, i.e., their intersection is empty and their union contains a path from v_{source} to v^s_{sink}. Finally, it contains $z_{\max}$ defined as the maximal recovery cost.

We can formulate the problem as follows:

$$\min \sum_{e \in E} c_e x_e + z_{\max}$$

subject to

$$\sum_{(k,q) \subseteq E \times E} z^s_{kq} = 1 \quad \text{for all } s \in S \tag{10}$$

$$x_e - \sum_{(k,q)\subseteq E\times E} a_{ek} z^s_{kq} = 0 \quad \text{for all } e \in E,\ s \in S \tag{11}$$

$$z_{\max} - \sum_{e\in E} f^s c_e \sum_{(k,q)\subseteq E\times E} a^s_{eq} z^s_{kq} \geq 0 \quad \text{for all } s \in S \tag{12}$$

$$x_e, \in \{0,1\} \quad \text{for all } e \in E \tag{13}$$

$$z^s_{kq}, \in \{0,1\} \quad \text{for all } k \subseteq E,\ q \subseteq E,\ s \in S, \tag{14}$$

where the index variables a_{ek} and a^s_{eq} are defined as follows:

$$a_{ek} = \begin{cases} 1 \text{ if edge } e \text{ is in edge set } k, \\ 0 \text{ otherwise.} \end{cases}$$

and

$$a^s_{eq} = \begin{cases} 1 \text{ if edge } e \text{ is in edge set } q \text{ for scenario } s, \\ 0 \text{ otherwise.} \end{cases}$$

Constraints (10) ensure that exactly one combination of initial and recovery edges is selected for each scenario; constraints (11) enforces that the same set of initial edges is selected for each scenario. Finally, constraints (12) make sure that $z_{\max}$ represents the cost of the worst case scenario.

Let λ, ρ_{es}, and π_s be the dual variables of the constraints (10), (11), and (12) respectively. The reduced cost of z^s_{kq} is now equal to:

$$c^{\text{red}}(z^s_{kq}) = -\lambda + \sum_{e\in E} \rho_{es} a_{ek} + \sum_{e\in E} \pi_s f^s c_e a^s_{eq}$$

Since we are dealing with a minimization problem, increasing the value of a variable improves the current LP-solution if and only if the reduced cost of this variable is negative. We have to solve the pricing problem for each scenario separately. For a given scenario s, the pricing problem amounts to minimizing $c^{\text{red}}(z^s_{kq})$ over all feasible a_{ek} and a^s_{eq}. This means that we have to select a subset of edges that contains a path from v_{source} to v^s_{sink}. This subset consists of edges which have been bought initially and edges which are attained during recovery. The first type corresponds to $a_{ek} = 1$ and has cost ρ_{es} and the second type to $a^s_{eq} = 1$ and has cost $\pi_s f^s c_e$. The pricing problem is close to a shortest path problem, but we have two binary decision variables for each edge. However, we can apply the following preprocessing steps:

- First, we select all edges with negative cost. From LP theory it follows that all dual variables π_s are nonnegative, and hence, all recovery edges have nonnegative cost. So we only select initial phase edges. From now on, the cost of these edges is considered to be 0.
- The other edges can either be selected in the initial phase or in the recovery phase. To minimize the reduced cost, we have to choose the cheapest option. This means that we can set the cost of an edge equal to $\min(\rho_{es}, \pi_s f^s c_e)$.

The pricing problem now boils down to a *shortest path* problem with nonnegative cost on the edges and hence can be solved by Dijkstra's algorithm [9].

5 Computational results

We performed extensive computational experiments with the knapsack problem. The algorithms were implemented in the Java Programming language and the Linear Programs were solved using ILOG CPLEX 11.0. All experiments were run on a PC with an Intel Core^{TM}2 Duo 6400 2.13GHz processor.

Our experiments were performed in three phases. In the first phase we tested 12 different instance types to find out which types are the most difficult. Our instance types are based on the instance types in [10], where we have to add the knapsack weight b_s and the probability p_s for each of the scenarios. In the second phase, we tested many different algorithms on relatively small instances. In the third phase we tested the best algorithms from the second phase on larger instances. In this section, we will present the most important issues from the second and third phase. We omit further details for reasons of brevity.

In the second phase we tested 5 instance classes, including subset sum instances. We considered instances with 5, 10, 15 and 25 items and with 2, 4, 6 and 8 scenarios. For each combination we generated 20 item sets and for each item set we generated 3 sets of scenarios, with large, middle, and small values of b_s relative to b, respectively. This means that we considered 4800 instances in total.

We report results on the following algorithms:

- Separate Recovery Decomposition with Branch-and-Price, where we branch on the fractional item with largest $\frac{c_j}{a_j}$ ratio and first evaluate the node which includes the item.
- Combined Recovery Decomposition with Branch-and-Price, where we branch in the same way as in Separate Recovery decomposition.
- Branch-and-Bound where we branch on the fractional item with smallest $\frac{c_j}{a_j}$ ratio and first evaluate the node which includes the item.
- Dynamic programming: a generalization of the DP solving the pricing problem in case of Combined Recovery Decomposition.
- Hill Climbing: we apply neighborhood search on the initial knapsack and compute for each initial knapsack the optimal recovery by Dynamic programming. Hill climbing performs 100 restarts.

For the branching algorithms we tested different branching strategies. In the branch-and-price algorithms the difference in performance turned out to be minor and we report on the strategy that performed best in Separate Recovery Decomposition. However, in the Branch-and-Bound algorithm some difference could be observed and we report on the strategy that shows the best performance for this algorithm.

The results of the second phase are given in Table 1. For each instance and each algorithm, we allowed at most 3000 milliseconds of computation time. For each algorithm, we report on the number of instances (out of 4800) that could not be solved within 3000 ms, the average and maximum computation time over the successful instances. For Hill Climbing we give the average and minimal

performance ratio and for the branching algorithms the average and maximum number of evaluated nodes.

Algorithm	Failed	avg t(ms)	max t(ms)	avg $\frac{c}{c^*}$	min $\frac{c}{c^*}$	avg nodes	max nodes
Separate Recovery	39	42	2,609	-	-	1.86	33
Combined Recovery	1,628	410	2,969	-	-	1.05	13
Branch and Bound	110	49	3,000	-	-	442.18	31,183
DP	3,466	401	2,953	-	-	-	-
Hill Climbing	0	40	1,516	1	0.88	-	-

Table 1. Second Phase Results

The results indicate that for this problem Separate Recovery Decomposition outperforms Combined Recovery Decomposition. DP is inferior to Branch-and-Bound and Hill Climbing.

In the third phase we experimented with larger instances for the two best algorithms. We present results for instances with 50 and 100 items and 2, 3, 4, 10, or 20 scenarios. Again, for each combination of number of items, number of scenarios, and instances class, we generated 20 item set with each 3 scenario sets. This results in 600 instances where the maximum computation time per instance per algorithm is 4 minutes. The results are depicted in Tables 2 and 3.

Items	Scenarios	Failed	avg t(ms)	max t(ms)	avg nodes	max nodes
50	2	4	354	35,031	1.32	31
50	3	5	1,050	51,032	1.5	27
50	4	22	1,141	39,999	1.42	21
50	10	53	812	36,421	1.11	25
50	20	59	287	50,515	1.01	3
100	2	88	1,151	42,468	1.24	39
100	3	114	388	28,844	1.18	19
100	4	107	175	1,000	1.04	9
100	10	97	287	1,204	1	1
100	20	96	370	1,125	1	1

Table 2. Third Phase Results for Separate Recovery decomposition

The results suggest that the computation time of Separate Recovery Decomposition scales very well with the number of scenarios. As may be expected, Hill Climbing shows a significant increase in the computation time when the number of scenarios is increased. Moreover, the small number of nodes indicates that Separate Recovery Decomposition is well-suited for instances with a larger number of scenarios. On average the quality of the solutions form Hill Climbing is very high. However, the minimum performance ratios of about 0.85 show that

Items	Scenarios	Failed	avg t(ms)	max t(ms)	avg $\frac{c}{c^*}$	min $\frac{c}{c^*}$
50	2	0	299	6,578	0.99	0.86
50	3	0	530	6,500	0.99	0.88
50	4	0	668	7,546	1	0.92
50	10	0	1,097	7,672	1	0.95
50	20	0	1,241	8,156	1	0.97
100	2	5	3,331	52,391	0.99	0.76
100	3	7	6,265	56,609	1	0.92
100	4	20	6,227	58,312	1	0.86
100	10	32	9,989	51,297	1	1
100	20	38	11,195	51,969	1	1

Table 3. Third Phase Results for Hill Climbing

there is no guarantee of quality. Observe that there is a difference in the notion of Failed. For the Separate Recovery Decomposition it means failed to solve to full optimality and for Hill Climbing failed complete the algorithm with 100 restarts. We did not yet include any primal heuristic in our Branch-and-Price algorithm. Including such a heuristic will enable us to make the notion of Failed similar to the one in Hill Climbing and is likely to decrease the number of failed instances.

6 Generalization and conclusion

In this paper we investigated column generation for recoverable robust optimization. We think that our approach is very promising and that it can be generalized to many other problems.

We presented two methods: Separate Recovery Decomposition and Combined Recovery Decomposition. In the first approach, we work with separate solutions for the initial problem and recovery solutions for the different scenarios; in the second one, we work with combined solutions for the initial problem and the recovery problem for a *single* scenario.

We considered the size robust knapsack problem. We applied Separate Recovery Decomposition and Combined Recovery Decomposition. In the first model, the pricing problem is a knapsack problem for both the initial solution columns and the recovery solution columns. In the second model, the pricing problem is to find an optimal column containing a combination of initial and recovery solution for a single scenario, i.e., recoverable robust optimization for a single scenario case. We implemented branch-and-price algorithms for both models. Our computational experiments revealed that for this problem Separate Recovery Decomposition outperformed Combined Recovery Decomposition and the first method scaled very well with the number of scenarios. If the algorithm is improved by a primal heuristic, it will find a feasible solution faster, which is able to reduce the number of Failed instances as reported in Table 2. This is an interesting topic for further research.

We developed a Combined Recovery model for the demand robust shortest path problem. We intend to implement this model in the near future. Interesting issues for further research are restrictions on the recovery solution such as a limited budget for the cost of the recovery solution or an upper bound on the number of edges obtained during recovery.

Finally, the generalization of the presented methods to other problems is a very interesting area for further research.

References

1. C. Barnhart, E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh, and P.H. Vance (1998). Branch-and-price: column generation for solving huge integer programs. *Operations Research 46*, 316–329.
2. A. Ben-Tal, L. El Ghaoui, and A. Nemirovski (2009). *Robust Optimization* Princeton University Press.
3. J.R. Birge, F. Louveaux (1997). *Introduction to Stochastic Programming*, Springer.
4. C. Büsing, A.M.C.A. Koster, and M. Kutschka (2010). *Recoverable Robust Knapsacks: the Discrete Scenario Case.* ftp://ftp.math.tu-berlin.de/pub/Preprints/combi/Report-018-2010.pdf
5. V. Cacchiani, A. Caprara, L. Galli, L. Kronn, G. Maroti, and P. Toth (2008) Recoverable Robustness for Railway Rolling Stock Planning. *8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS 2008).* http://drops.dagstuhl.de/opus/volltexte/2008/1590/
6. A. Caprara, L. Galli, L.G. Kroon, G. Maróti, P. Toth: Robust Train Routing and Online Re-scheduling. *ATMOS 2010* http://drops.dagstuhl.de/opus/volltexte/2010/2747/
7. S. Cicerone, G. D'angelo, G. Di Stefano,, D. Frigioni, A. Navarra, M. Schachtebeck and A. Schöbel (2009) Recoverable Robustness in Shunting and Timetabling . In R.K. Ahula, R.H. Möhring and C.D. Zaroliagas (eds.), *Robust and On-Line Large Scale Optimization.* Lecture Notes in Computer Science, Vol. 5868, 28–60. Springer.
8. K. Dhamdhere, V. Goyal, R. Ravi, and M. Singh (2005). How to pay, come what may: Approximation algorithms for demand-robust covering problems. *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science (FOCS '05).* pp. 367-378. IEEE Computer Society.
9. E.W. Dijkstra (1959). A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 269–271.
10. H. Kellerer, U. Pferschy, and D. Pisinger (2004). *Knapsack Problems.* Springer, Berlin, Germany.
11. C. Liebchen, M. Lübbecke, R.H. Möhring, and S. Stiller (2009). Recoverable Robustness. In R.K. Ahula, R.H. Möhring and C.D. Zaroliagas (eds.), *Robust and On-Line Large Scale Optimization.* Lecture Notes in Computer Science, Vol. 5868, 1–27, Springer.
12. . C. Puhl. Recoverable robust shortest path problems. http://www.di.unipi.it/optimize/Events/proceedings/T/B/2/TB2-2.pdf