Decomposition approaches for recoverable robust optimization problems

J.M. van den Akker
P.C. Bouman
J.A. Hoogeveen
D.D. Tönissen

Technical Report <Technical Report number>
September 2014
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

Decomposition approaches for recoverable robust optimization problems

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

[a] *Department of Information and Computing Sciences Utrecht University, Princetonplein 5, 3584 CC Utrecht, The Netherlands*

[b] *Rotterdam School of Management Erasmus University, Burgemeester Oudlaan 50, 3062 PA Rotterdam, The Netherlands*

[c] *School of Industrial Engineering Eindhoven University of Technology, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands*

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 possible disturbances by including recourse actions in your planning strategy. Well-known approaches to create plans that take possible, common disturbances into account are robust optimization and stochastic programming. More recently, another approach has been developed that combines the best of these two: recoverable robustness. In this paper, we solve recoverable robust optimization problems by the technique of column generation. We consider two types of decomposition approaches: separate recovery and combined recovery. We investigate our approach for 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. For each problem, we present elaborate computational experiments. We think that our approach is very promising and can be generalized to many other problems.

Keywords: column generation; robustness; knapsack; shortest path

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,

*Correspondig author

Email addresses: J.M.vandenAkker@uu.nl (J.M. van den Akker), PBouman@rsm.nl (P.C. Bouman), J.A.Hoogeveen@uu.nl (J.A. Hoogeveen), D.D.Tonissen@tue.nl (D.D. Tönissen)

[1] The research was performed while this author was at Utrecht University

different approaches have been developed. In case of *robust optimization* (see [2], [3]) we choose the solution with minimum cost that remains feasible for a given set of disturbances in the parameters. In case of *stochastic programming* [4], we take *first stage decisions* on basis of the current information and, after the true value of the uncertain 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 [4]. Summarized, 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* [17] 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 set of pre-described, fast, and simple recovery algorithms. 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 this approach 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 hours 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. This may lead to a very conservative schedule. 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 recoverable robust optimization methods have gained a lot of attention (see e.g. [10], [11])

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. We consider two types of solution approaches: Separate Recovery and Combined Recovery. Our approach can be generalized to many other problems. This paper extends our conference paper [5] by presenting a general definition of our decomposition approaches and a further study of the solution algorithm for the demand robust shortest path problem. This study includes different column generation strategies and elaborate computational experiments. To the best of our knowledge, [5]

and this paper are the first ones applying column generation to recoverable robust optimization. Another decomposition approach, namely Benders decomposition, is used in [9] 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 present our two different decomposition approaches. In Section 4, we consider the size robust knapsack problem. We investigate the two decomposition approaches and we present computational experiments in which we compare different solution algorithms. Besides algorithms based on Separate and Combined Recovery Decomposition, we test hill-climbing, dynamic programming and branch-and-bound. The experiments indicate that Separate Recovery Decomposition performs best. Section 5 is devoted to the demand robust shortest path problem. Since Separate Recovery Decomposition does not seem to be appropriate for this problem, we focus on Combined Recovery Decomposition and consider the settings of the branch-and-price algorithm in more detail. In our experiments we show that the column generation strategy has a significant influence on the computation time. 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 discrete 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:

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

Here, $c^s(y^s)$ denotes the cost associated with the recovery variables y^s and g the function to combine these cost into the objective function. 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 maximum function, i.e., it models the maximal cost of the recovered solutions y^s. This corresponds to minimizing the worst-case cost. If $c^s(y^s)$ measures the deviation of the solution y^s from x, we minimize the maximum deviation from the initial solution. Note that this deviation may also be limited by the recovery algorithms.
3. Suppose we are given the probabilities p_s of scenarios s. Then g can be defined as the expected value of the solution after recovery, i.e., $\sum_{s\in S} p_s c^s(y^s)$.

Although earlier papers on recoverable robustness (e.g. [17]) 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. Decomposition approaches

We discuss two decomposition approaches for recovery robust optimization problems. In both cases we reformulate the problem such that we have to select one solution for the initial problem and one for each scenario. The difference consists of the way we deal with the scenarios.

3.1. Separate Recovery Decomposition

In *Separate Recovery Decomposition*, we select an initial solution and *separately* we select a solution for each scenario. This means that for each feasible initial solution $k \in F$ we have a decision variable x_k signalling if this solution is selected; similarly for each feasible solution for each scenario $q \in F_s$ we have a decision variable y^s_q. In the formulation we enforce that we select exactly one initial solution and one solution for each scenario. We also enforce that for each scenario the initial solution can be transformed into a feasible solution by the given recovery algorithm. We assume that the recovery constraint and the objective function can be expressed *linearly*. We now obtain an Integer Linear Programming formulation which is formulated as follows (for maximization objective):

$$\max f(\{x_k | k \in F\}) + g(\{c^s(\{y^s_q | q \in F_s\}) | s \in S\})$$

subject to

$$\sum_{k\in F} x_k = 1 \tag{1}$$

$$\sum_{q\in F_s} y^s_q = 1 \quad \text{for all } s \in S \tag{2}$$

$$A_1 x + A^s_2 y^s \leq b \quad \text{for all } s \in S \tag{3}$$

$$x_k \in \{0,1\} \quad \text{for all } k \in F \tag{4}$$

$$y_q^s \in \{0,1\} \quad \text{for all } q \in F_s,\ s \in S \tag{5}$$

Here y^s denotes the vector of all entries y_q^s. 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 $y_q^s \geq 0$, and solve this LP-relaxation. To deal with the large number of variables we are going to solve the problem by *column generation*.

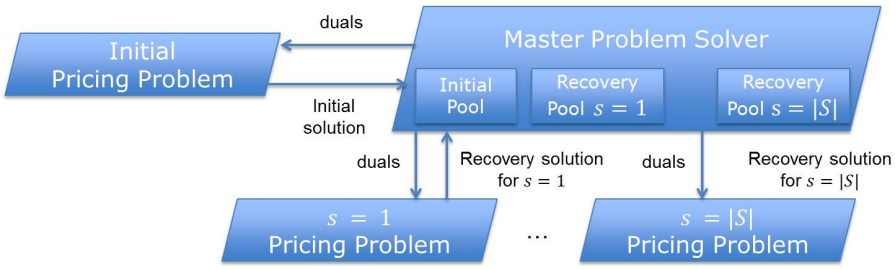

Figure 1: Column generation process Separate Recovery Decomposition

The column generation process is depicted in Figure 1. We start with a limited subset of the variables and solve the LP-relaxation for this subset only; this is called the Restricted Master Problem. Then we solve the Pricing Problem, i.e., we look for variables that are not yet included in the Restricted Master Problem and can improve the solution if their value is made positive. For variables x_k this boils down to an optimization problem over F, i.e. a variant of the initial problem, and for variables y_q^s for scenario s to an optimization problem over F_s, i.e. a variant of the original problem in case of scenario s. If improving variables are found, they are added to the Restricted Master Problem, it is solved again, after which pricing is performed etc. If pricing does not find any new variables we know that the Master Problem has been solved to optimality. To find an integral solution, we are going to apply Branch-and-Price, i.e., combine column generation with Branch-and-Bound, where the solution value of the LP is used as an upper bound.

3.2. Combined Recovery Decomposition

In *Combined Recovery Decomposition*, we select for each scenario a combination of an initial solution together with the optimal recovery a solution for that single scenario. This means that for each scenario $s \in S$ we have for each combination of an initial solution $k \in F$ and the corresponding solution $q \in F_s$ obtained by the recovery algorithm a binary variable z_{kq}^s signalling if this combination is selected. We enforce that only one combination will get selected for each scenario in the master problem. Moreover, it is important to ensure that the combinations selected for the different scenarios all correspond to the same initial solution. We assume that the functions

f and g can be expressed in a *linear* way. We obtain the following Integer Linear Programming formulation:

$$\max f(x) + g(\{c^s(\{z^s_{kq} |\ k \in F,\ q \in F_s, q = A(k,s)\}) | s \in S\})$$

subject to

$$\sum_{(k,q)\in F\times F_s} z^s_{kq} = 1 \quad \text{for all } s \in S \tag{6}$$

$$x = Az^s \quad \text{for all } s \in S \tag{7}$$

$$z^s_{kq} \in \{0,1\} \quad \text{for all } k \in F,\ q \in F_s,\ q = A(k,s),\ s \in S, \tag{8}$$

Here z^s denotes the vector of all entries z^s_{kq}. We also solve this ILP formulation by Branch-and-Price. The column generation process is depicted in Figure 2. The Pricing Problem now boils down to finding an optimal combination of an initial solution and a recovery solution for a given scenario.

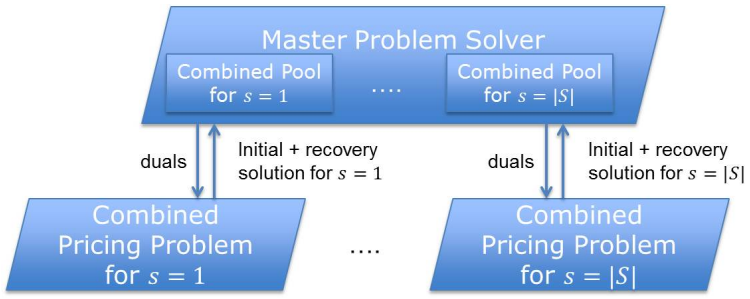

Figure 2: Column generation process Combined Recovery Decomposition

We think that our approach can be applied to different recoverable robust optimization problems. In the next two sections, we will demonstrate the approach for two different problems.

4. 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. Our objective is to maximize the expected revenue after recovery. We study the situation in which recovery has to be performed by removing items. As soon as it becomes clear which scenario s appears, recovery is performed by removing items in such a way that the remaining items give a knapsack with maximal revenue and size at most b_s. This boils

down to solving a knapsack problem where the item set is the set of items selected in the initial solution and the knapsack size is b_s. Hence, our set of recovery algorithms is given by the dynamic programming algorithm for solving these knapsacks.

Recently [7] have studied a different version of our knapsack problem, where they focus on uncertainty in the weights. They show $\mathcal{NP}$-hardness of several variants of the problem and develop a polyhedral approach to solve these problems. A follow-up paper [8] presents an integer linear programming formulation of quadratic size and evaluates the gain of recovery. There is also some research with a focus on approximation algorithms. Disser et al [14] consider policies for packing a knapsack with unknown capacity and Goerigk et al [15] consider the knapsack problem in which there is a limited budget to decrease item weights.

In this section, we are going to discuss our two decomposition approaches for the size robust knapsack problem and present elaborate computational experiments in which we compare our method with other algorithms.

4.1. 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 $q \in K(b_s)$ by $C_q^s = \sum_{i \in q} c_i$.

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

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

The problem can now be formulated as follows.

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

subject to

$$\sum_{k \in K(b)} x_k = 1 \tag{9}$$

$$\sum_{q \in K(b_s)} y_q^s = 1 \quad \text{for all } s \in S \tag{10}$$

$$\sum_{k \in K(b)} a_{ik} x_k - \sum_{q \in K(b_s)} a_{iq}^s y_q^s \geq 0 \quad \text{for all } i \in \{1, 2, \ldots, n\},\ s \in S \tag{11}$$

$$x_k \in \{0, 1\} \quad \text{for all } k \in K(b) \tag{12}$$

$$y_q^s \in \{0, 1\} \quad \text{for all } q \in K(b_s),\ s \in S, \tag{13}$$

where the index variables a_{ik} and a^s_{iq} 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

$$a^s_{iq} = \begin{cases} 1 & \text{if item } i \text{ is in knapsack } q \in K(b_s), \\ 0 & \text{otherwise.} \end{cases}$$

In the above model constraint (9) states that exactly one knapsack is selected for the original situation and constraints (10) that exactly one knapsack is selected for each scenario. Constraints (11) ensures that recovery is done by removing items.

Recall that we solve this ILP formulation using Branch-and-Price, we relax the integrality constraints (12) and (13) into $x_k \geq 0$ and $y^s_k \geq 0$, and solve this LP-relaxation by column generation.

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 (9), (10), and (11) 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 $c^{\text{red}}(y^s_q)$ are given by $c^{\text{red}}(y^s_q) = \sum_{i=1}^{n} a^s_{iq}(p_s c_i + \pi_{is}) - \mu_s$ 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, i.e. items i for which $\sum_{k \in K(b)} a_{ik} x_k$ is fractional. This is easily combined with column generation, since enforcing that an item is taken in or omitted from the knapsack, can easily be included in the pricing problem. If all values $\sum_{k \in K(b)} a_{ik} x_k$ are integral, then a single initial knapsack is selected with value 1. Now consider a basic solution to the LP for

scenario s. It is easy to see that this solution contains an optimal subset of the initial knapsack with total weight at most b_s which is selected with value 1. Consequently, the solution is integral.

4.2. 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:

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

Clearly, z^s_{kq} is only defined if q is a subset of k. 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:

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

subject to

$$\sum_{(k,q) \in K(b) \times K(b_s)} z^s_{kq} = 1 \quad \text{for all } s \in S \tag{14}$$

$$x_i - \sum_{(k,q) \in K(b) \times K(b_s)} a_{ik} z^s_{kq} = 0 \quad \text{for all } i \in \{1, 2, \ldots, n\},\ s \in S \tag{15}$$

$$x_i, \in \{0,1\} \quad \text{for all } i \in \{1, 2, \ldots, n\} \tag{16}$$

$$z^s_{kq}, \in \{0,1\} \quad \text{for all } k \in K(b),\ q \in K(b_s),\ s \in S, \tag{17}$$

Constraints (14) enforce that exactly one combination is selected for each scenario; constraints (15) 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^s_{kq}. We denote the dual variables of constraints (14) and (15) by ρ_s and σ_{is}, respectively. The reduced cost of z^s_{kq} is now equal to:

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

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:

$$\begin{aligned} D(i,0,0) &= 0 \quad \forall i \\ D(0,w_0,w_s) &= -\infty \quad \text{for } w_0, w_s > 0 \\ D(i,w_0,w_s) &= \max \begin{cases} D(i-1,w_0,w_s) \\ D(i-1,w_0-a_i,w_s)+\sigma_{is} \\ D(i-1,w_0-a_i,w_s-a_i)+\sigma_{is}+p_s c_i \end{cases} \end{aligned}$$

4.3. 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™Duo 2.13 GHz processor.

Our experiments were performed in three phases. Since we want to focus on difficult instances, 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 [16], 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 observations 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 (except for 5 items were we only considered 2 and 4 scenarios). For each combination we generated 100 item sets (20 from each instance class) 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 4200 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.

Algorithm	Failed	avg t(ms)	max t(ms)	avg $\frac{c}{c*}$	min $\frac{c}{c*}$	avg nodes	max nodes
Separate Recovery	128	107	2563	-	-	3.27	122
Combined Recovery	1407	417	2969	-	-	1.12	17
Branch and Bound	190	111	2906	-	-	1281	33321
DP	2840	347	2984	-	-	-	-
Hill Climbing	0	17.3	422	0.99	0.85	-	-

Table 1: Second Phase Results

- 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 4200) 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. For Hill Climbing 'Failed' means that it was not able to finish all restarts in the given time.

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, we generated 100 item sets (20 from each instance class) with each 3 scenario sets. This results in 300 instances per combination of number of

items and number of scenarios, 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 ms	max ms	avg nodes	max nodes
50	2	2	686	56312	1.56	68
50	3	12	2724	53454	1.7	25
50	4	46	3799	58688	2.6	35
50	10	125	3295	53483	2.29	35
50	20	144	1473	38766	1.4	17
100	2	114	1695	47531	1.05	5
100	3	173	703	24781	1.16	11
100	4	176	964	46172	2.03	59
100	10	213	469	34547	1.39	25
100	20	210	103	2703	1.13	13

Table 2: Third Phase Results for Separate Recovery decomposition

Items	Scenarios	Failed	avg ms	max ms	avg $\frac{c}{c^*}$	min $\frac{c}{c^*}$
50	2	0	104	969	0.98	0.68
50	3	0	173	1204	0.98	0.84
50	4	0	180	1203	0.98	0.83
50	10	0	268	1407	1	0.94
50	20	0	309	1515	1	0.84
100	2	0	887	19656	0.98	0.66
100	3	0	1257	25578	1	0.86
100	4	0	1783	32625	1	0.8
100	10	0	3546	34703	1	0.8
100	20	0	4546	37312	1	0.94

Table 3: Third Phase Results for Hill Climbing

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 from Hill Climbing is very high. However, the minimum performance ratios of about 0.66 show that 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.

5. The demand robust shortest path problem

The demand robust shortest path problem is an extension of the shortest path problem and has been introduced in [12]. 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. To work with the same problem as [12], we choose as objective to minimize the cost of the worst case scenario. It is not difficult to see that this problem is $\mathcal{NP}$-hard, as it generalises the Steiner Tree Problem. When we pick each f^s high enough, the optimal solution is to buy a minimum cost tree that connects the source and all sinks during the first phase.

In contrast to [6], 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. 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. Hence, the recovery algorithm is a shortest path algorithm.

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}. It is quite involved to express this constraint using linear inequalities, and hence to apply Separate Recovery Decomposition. However, the constraint fits very well into 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:

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

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

$$\sum_{(k,q) \subseteq E \times E} z^s_{kq} = 1 \quad \text{for all } s \in S \tag{18}$$

$$x_e - \sum_{(k,q) \subseteq E \times E} a_{ek} z^s_{kq} = 0 \quad \text{for all } e \in E,\ s \in S \tag{19}$$

$$z_{\max} - \sum_{e \in E} f^s c_e \sum_{(k,q) \subseteq E \times E} a^s_{eq} z^s_{kq} \geq 0 \quad \text{for all } s \in S \tag{20}$$

$$x_e, \in \{0,1\} \quad \text{for all } e \in E \tag{21}$$

$$z^s_{kq}, \in \{0,1\} \quad \text{for all } k \subseteq E,\ q \subseteq E,\ s \in S, \tag{22}$$

where the binary index variables a_{ek} signal if edge e is in edge set k and the binary index variables a^s_{eq} signal if edge e is in edge set q for scenario s.

Constraints (18) ensure that exactly one combination of initial and recovery edges is selected for each scenario; constraints (19) enforces that the same set of initial edges is selected for each scenario. Finally, constraints (20) make sure that $z_{\max}$ represents the cost of the worst case scenario.

5.1. Solving the LP by column generation

We first relax the integrality constraints (21) and (22) into $x_e \geq 0$ and $z^s_{kq} \geq 0$, and solve this LP-relaxation. To deal with the huge number of variables we are going to solve the problem by *column generation*.

The pricing problem

In this case, the pricing problem then boils down to minimizing the reduced cost.

Let λ_s, ρ_{es}, and π_s be the dual variables of the constraints (18), (19), and (20) respectively. The reduced cost of z^s_{kq} is now equal to:

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

We have to solve the pricing problem for each scenario *separately*. For a given scenario s, the pricing problem boils down 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*:

1. 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 with negative cost ρ_{es}. From now on, the cost of these edges is considered to be 0.
2. 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. We implemented the algorithm by a min heap with running time $O(|E| \log(|V|))$.

Since we solve the pricing problem for each scenario separately, the following questions arise: "For which scenarios do we actually solve the pricing problem?" and "Which columns do we actually add to the restricted LP?". We investigate the following strategies:

- *Interleaved:* goes through the pricing problems of the different scenarios one by one. As soon as a variable with negative reduced cost is identified, the corresponding column is added and the master problem gets resolved. After that, it goes to the next scenario. When the pricing problem has a solution with nonnegative reduced cost for every scenario the column generation process is stopped.
- *Best:* Solves the pricing problem for all scenarios, but only a column z^s_{kq} with overall minimal reduced cost is added to the master problem. The master problem is solved again and this repeats itself until the minimal reduced cost is nonnegative.
- *All:* Solves the pricing problem for all scenarios and adds a column for all scenarios for which a variable z^s_{kq} with negative reduced cost was found, after adding all those columns it resolves the master problem.

Within the first few experiments it became very clear that the LP problem is very degenerate. Certainly for larger graphs with a lot of scenarios this tends to slow down the computation enormously. Observe that every solution needs at least $|S|$ columns. To get a complete solution, because of constraint (19), we need a collection of columns such that for each edge e the total amount by which it is selected in the initial solution $\sum_{(k,q)\subseteq E\times E} a_{ek} z^s_{kq}$ is the same for every scenario s. This has the consequence that, although it is included in the basis, a promising new column often does not influence the primal solution. To deal with this problem, we use the following method: When

a column is added, we always guarantee that it can be selected for the solution by generating for every scenario the best column with the same initial edges. Those columns are generated by fixing the set of initial edges and finding the best recovery edges by running Dijkstra for all $|S|-1$ scenarios.

As a starting solution we take the column in which all edges are taken in the initial solution. Other strategies were tested but the differences were small and instance dependent.

Moreover, we have investigated column deletion, i.e. deletion of columns with too positive reduced cost. However, this does not seem to work well in combination with including additional columns.

5.2. Solving the ILP

If the solution of the LP-relaxation is integral, our problem is solved to optimality. Otherwise, we proceed by *Branch-and-Price* [1], i.e. Branch-and-Bound, where we generate additional column in the nodes of the search tree.

In a Branch-and-Price algorithm the branching strategy has to be designed in such a way that we are still able to solve the pricing problem in each node of the tree. In our algorithm we branch on the variables x_e. In a node with $x_e = 1$ we only generate columns where edge e is bought in the initial phase. This implies that in the first preprocessing step of the pricing we buy edge e at cost ρ_{es} and then set its cost to 0. In a node with $x_e = 0$ we are not allowed to buy edge e in the initial phase. Therefore, we have to define the cost of the edge as $\pi_s f^s c_e$ instead of $\min(\rho_{es}\pi_s f^s c_e)$.

Concerning the choice of the edge for branching, besides considering the edges in order of their index, we implemented branching on the most doubtful edge. This means that the we branch on the edge for which $|x_e - \frac{1}{2}|$ is minimal. This strongly speeds up the computation.

Moreover, we investigated different node selection strategies. We considered best bound branching, i.e. branching on the node with the minimal lower bound, breadth first search, depth first search and also best depth first, which from the deepest nodes in the tree selects the one with the best lower bound. In our experiments best depth first did not improve depth first search very much. Although depth first search sometimes slightly improved best bound search, it showed a less stable behaviour. The same is true for breadth first search. Therefore we chose to use best bound branching in our algorithm.

To compute an upper bound three rounding heuristics were tested. The first heuristic was to select for the initial phase only edges with $x_e = 1$ in the LP-solution. In the second heuristic, all edges with $x_e \geq \frac{1}{2}$ were selected in the initial solution. As a third alternative we applied a randomized strategy: each edge was selected in the initial solution with a probability equal to the value of x_e in the optimal solution of the LP-relaxation. In all three cases, for each scenario the best recovery

solution was determined by Dijkstra's algorithm [13]. There did not seem to be much difference in performance between the heuristics and we applied the second one since we thought it to be the most intuitive one.

5.3. Computational results

We have implemented our column generation and branch-and-price algorithms in Java and used ILOG CPLEX 12.4 as linear programming solver. We ran experiments on an Intel®Core™Duo 2.66 GHz processor.

Again, our experiments were performed in three phases. We first investigated all column generation strategies, to determine the best one. Secondly, we performed a sensitivity analysis. Finally, we ran our algorithm on some larger instances.

We first present results for linear programming to illustrate the effect of the different column addition strategies from Section 5.1. The strategies InterA, BestA, and AllA denote extensions of the strategies Inter, Best and All, in which, when we add a column, we also add for each scenario the best column with the same initial edges. In Table 4 we give results for 4 different relatively small instances, where $G_{n,e}$ has a graph with n nodes and e edges. The recovery factor f for these instances is fixed at 2.0 and every non source node is a possible sink and thus a scenario. For each instance, we give the number of iterations (it),the number of added columns (col) and the computation time in milliseconds (t).

	$G_{4,5}$			$G_{14,13}$			$G_{17,23}$			$G_{17,31}$		
Method	it	col	t	it	col	t	it	col	t	it	col	t
Inter	32	40	10	613	639	1014	3701	3740	25.941	7132	7179	134213
Best	22	30	10	545	571	1038	2835	2874	19.278	6065	6112	136585
All	17	39	8	114	699	554	577	3632	8069	1101	7453	54145
InterA	5	21	6	33	443	55	64	1048	393	537	8624	13132
BestA	6	24	4	24	326	39	73	1192	583	257	4144	7503
AllA	8	21	3	40	443	55	81	1048	407	652	8624	16649

Table 4: Results for linear programming

Our results reveal that the strategies with additional columns strongly speed up the computation. In most of our cases the number of columns is also reduced, but as may be expected, the reduction is not that strong.

We also solved the ILP for these instances, where we applied all combinations of strategies in the root and in the tree. The strategies without additional columns resulted in large running times.

Moreover, it did not pay off to use a different strategy in the root than in the remainder of the tree. Therefore, for Integer Linear Programming, we only consider strategies with additional columns and use the same strategy for the complete tree.

Recall that we branch on the most doubtful edge with $|x_e - \frac{1}{2}|$ minimal and select the node with the best lower bound. We first report results for a set $\mathcal{G}_{500}$ of 500 random instances. They are based on graphs with 10 to 29 edges, where for each number of edges we vary the number of nodes. For every number of edges a total of 25 graphs are generated. All graphs are connected, the cost of the edges uniformly random from the interval $[0; 100]$, every non-source is a possible sink, and f has a random value in the interval $[1; 10]$.

The average solution times for those instances are 53.6, 21.2, 66.7 and 34.8 seconds for InterA, BestA, and AllA and BestANoSort, where in the latter strategy we branch on the edges in order of their numbering instead of on the most doubtful edge. The BestA method performs significantly better than the other methods according the Wilcoxon signed-rank test (done with R version 3.1.1, with $p = 3.823e^{-13}$ as the highest p). Table 5 shows results for a subset of the set of random instances $\mathcal{G}_{500}$. For each number of edges and each strategy, we report on the average total number of iterations of column generation (it), the average number of nodes in the branch-and-bound tree (nodes), and the average computation time in milliseconds (t).

	InterA			BestA			BestANoSort			AllA		
ed	it	nd	t	it	nd	t	it	nd	t	it	nd	t
11	37	1	20	30	1	19	36	2	24	49	1	20
14	153	5	135	104	6	129	143	10	158	195	5	139
17	385	11	791	209	10	510	358	19	689	472	11	824
20	715	17	2842	414	19	1510	1042	77	2417	888	17	2951
23	2493	41	24669	1353	49	10208	2782	119	14832	3016	41	26606
26	10121	157	174213	4520	154	64500	18514	1238	119645	11921	157	193029
29	20630	325	488442	9641	256	184075	21837	638	297424	23939	272	571894

Table 5: Results for branch-and price with random instances

In Figure 3 we plot on a logarithmic scale the computation time for each number of edges.

These results suggest that especially for larger graphs BestA outperforms the other column addition strategies. Even when BestA is combined with the inferior branching strategy of branching on edges in lexicographical order, this is faster than the other column addition strategies.

We also did some sensitivity experiments on the influence of the edge cost and the recovery factor f. To test the influence of the costs of the edges we used $G_{14,13}$ and $G_{17,31}$ with fixed recovery

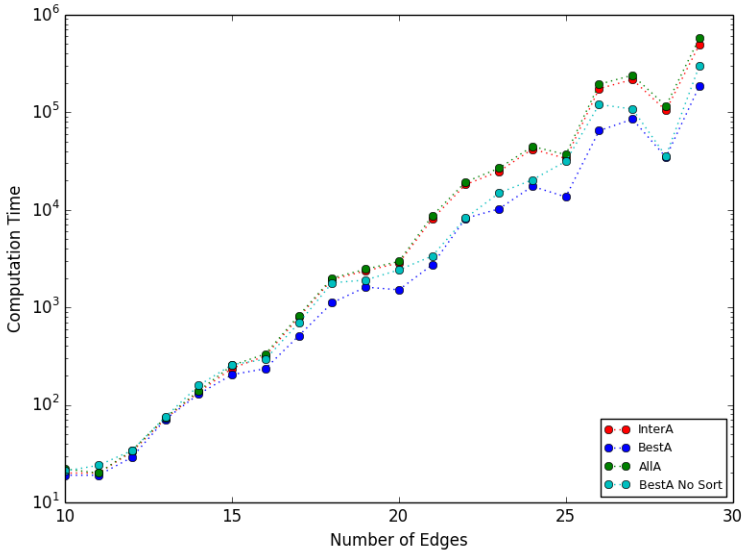

Figure 3: Logarithmic time results of the methods.

factor $f = 2$. We created 500 random cost versions by generating the edge costs uniformly random from the interval $[0; 50]$. For the recovery factor f we generated similar instances only now we fixed the cost and varied the recovery factor in the interval $[1; 10]$. Because the $G_{17,31}$ graph with random recovery factor f was solved relatively slow, we only solved 25 instances. These are solved with the BestA method and the results of these instances can be found in Table 6.

	$G_{14,13}$ c			$G_{17,31}$ c			$G_{14,13}$ f			$G_{17,31}$ f		
	it	nd	t	it	nd	t	it	nd	t	it	nd	t
best	30	3	57	146	3	281	27	1	161	483	1	42651
25%	134	13	235	1044	38	6323	51	1	325	3183	39	100754
median	185	29	325	1922	85	13289	74	1	416	6964	99	173167
75%	235	49	445	3177	166	18687	116	9	606	9404	153	238108
worst	675	211	1475	9753	1567	51236	279	73	1112	51923	1221	520106

Table 6: Results for varying the cost (c) and the recovery factor (f) of the edges

These results suggest that cost as well as recovery factor have a large influence on the solution time of the instance, this different can be a factor of more than 100. This might be explained by the fact that some combinations of cost and recovery factor result in alternative solutions with approximately the same value, which have an impact on the size of the search tree. We consider a small example with 3 nodes: one source s and two possible sink t_1 and t_2 each occurring with probability $\frac{1}{2}$. There are two edges (s, t_1) and (s, t_2) with the same initial cost Q. If $f = 2$, then it does not make a difference if you buy all edges, one edge, or no edges in the initial phase.

The experiments from now on, were performed with a better computer with an Intel®Core™i5 3.40 GHz processor. This computer is approximately twice as fast. Until now we considered instances were every non-source node could be the sink, which are instances with a relatively large number of scenarios. Since the size of the ILP model is linear in the number of scenarios, we may expect that instances with fewer scenarios can be solved faster. From the set $\mathcal{G}_{500}$ used before we generated 42747 new random instances by varying the number of scenarios. These were solved with the BestA method. Solving all 42747 instances took 56.4 hours. In Table 7 we show the average computation time in milliseconds for different numbers of edges and different numbers of scenarios.

	number of scenarios			
edges	5	10	15	20
11	9	-	-	-
14	19	123	-	-
23	102	1390	7103	-
26	120	3495	19941	59429
29	573	16296	107511	249716

Table 7: Results different number of scenarios

Moreover, in Figures 4 and 5, we plot the computation time on a logarithmic scale, per number of edges as a function of the number of scenarios and per number of scenarios as a function of the number of edges, respectively. A larger version of the figures can be found in the appendix.

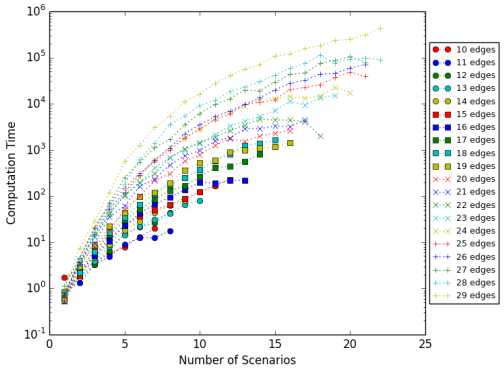

Figure 4: Time results per number of edges

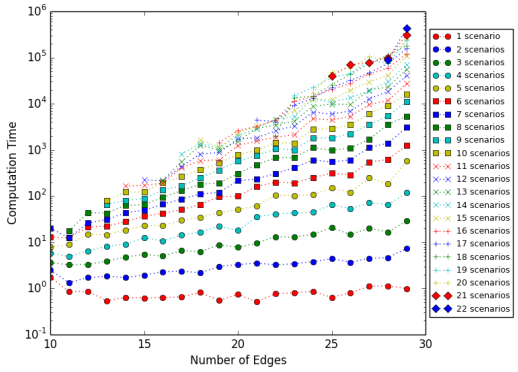

Figure 5: Time results per number of scenarios

Our results suggest that the number of edges has a strong impact on the computation time, our graph indicates exponential behaviour. The impact of the number of scenarios, seems very strong in the beginning but then somewhat flattens out in the logarithmic scale. We conclude that both

have a strong influence.

Finally, we performed experiments for a few larger instances, with the maximum number of scenarios: every non source node is a possible sink. In Table 8 we show the number of nodes and edges, the time to solve the LP, the time to solve the ILP, and the total solution time, and the number of nodes in the branch-and-bound tree, together with the node in which the best solution was found.

Graph	$\|V\|$	$\|E\|$	t_{LP}	t_{ILP}	t	n	n_{sol}
G_1	13	50	3.6 min	3.6 min	7.2 min	3	2
G_2	25	50	19.3 min	2083 min	35 hr	1597	1332
G_3	15	60	27.2 min	167 min	3.2 hr	207	142
G_4	30	60	167 min	-	-	-	-

Table 8: Results for large instances

6. Conclusion

In this paper we investigated column generation for recoverable robust optimization. We think that our approach is very promising and that it might be applicable 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 an initial and a 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. We also tested a few other methods, such as a branch-and-bound and a hill-climbing algorithm. Separate Recovery Decomposition outperformed the branch-and-bound algorithm. The hill-climbing algorithm provided very good solutions but no performance guarantee (for one instances we observed a performance ratio of 0.66). If we improve the primal heuristic in the Separate Recovery

Decomposition algorithm, it will find a feasible solution faster, which is able to reduce the computation time and in this way the number of Failed instances as reported in Table 2. This is an interesting topic for further research.

We presented a Combined Recovery Decomposition model for the demand robust shortest path problem. The pricing problem boils down to a shortest path problem. We developed and tested a branch-and-price algorithm to solve this problem. The column addition strategy turns out to be extremely important. In particular, the algorithm is strongly improved when we include additional columns in the following way. Whenever a column with negative reduced cost is added, also for each scenario the column with the same initial solution and the best recovery solution is added. The most effective strategy is the one we call 'BestA'. This strategy finds the minimum reduced cost column for each scenario, adds the single column with the most negative reduced cost over all scenarios, and then includes additional columns as described above.

The solution time is very sensitive to the cost of the edges and the recovery factor, this makes it unfortunately hard to predict how long it takes to solve a specific instance. By solving a large amount of graphs insight on the influence of the amount of edges and scenarios is gained. For our instances the number of edges seems to have an exponential influence on the time.

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, we think that our approach can be generalized to solve many other problems. We are currently investigating our framework for different applications.

References

[1] Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, and Pamela H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.

[2] A. Ben-Tal, L. El Ghaoui, and A. Nemirovski. *Robust Optimization.* Princeton University Press, 2009.

[3] D. Bertsimas and M. Sim. The price of robustness. *Operations Research*, 52(1):35–53, 2004.

[4] John R. Birge and François Louveaux. *Introduction to Stochastic Programming.* Springer Series in Operations Research and Financial Engineering. Springer, 1997.

[5] P.C. Bouman, J.M. van den Akker, and J.A. Hoogeveen. Recoverable robustness by column generation. In Camil Demetrescu and Magnús M. Halldorsson, editors, *Algorithms ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 215–226. Springer Berlin Heidelberg, 2011.

[6] Christina Büsing. Recoverable robust shortest path problems. *Networks*, 59(1):181–189, 2012.

[7] Christina Büsing, Arie Koster, and Manuel Kutschka. Recoverable robust knapsacks: the discrete scenario case. *Optimization Letters*, pages 1–14, 2011.

[8] Christina Büsing, Arie M.C.A. Koster, and Manuel Kutschka. Recoverable robust knapsacks: γ-scenarios. In Julia Pahl, Torsten Reiners, and Stefan Voß, editors, *Network Optimization*, volume 6701 of *Lecture Notes in Computer Science*, pages 583–588. Springer Berlin Heidelberg, 2011.

[9] Valentina Cacchiani, Alberto Caprara, Laura Galli, Leo G. Kroon, and Gábor Maróti. Recoverable robustness for railway rolling stock planning. In *ATMOS 2008 - 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems*, Karlsruhe, Germany, 2008.

[10] Alberto Caprara, Laura Galli, Leo G. Kroon, Gábor Maróti, and Paolo Toth. Robust train routing and online re-scheduling. In *ATMOS 2010 - 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems*, pages 24–33, Liverpool, United Kingdom, 2010.

[11] Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, Daniele Frigioni, Alfredo Navarra, Michael Schachtebeck, and Anita Schöbel. Recoverable robustness in shunting and timetabling. In *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*, pages 28–60. 2009.

[12] Kedar Dhamdhere, Vineet Goyal, R. Ravi, and Mohit Singh. How to pay, come what may: Approximation algorithms for demand-robust covering problems. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '05, pages 367–378, Washington, DC, USA, 2005. IEEE Computer Society.

[13] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] Yann Disser, Max Klimm, Nicole Megow, and Sebastian Stiller. Packing a Knapsack of Unknown Capacity. In Ernst W. Mayr and Natacha Portier, editors, *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 276–287, Dagstuhl, Germany, 2014.

[15] Marc Goerigk, Yogish Sabharwal, Anita Schöbel, and Sandeep Sen. Approximation algorithms for the weight-reducible knapsack problem. In T.V. Gopal, Manindra Agrawal, Angsheng Li, and S.Barry Cooper, editors, *Theory and Applications of Models of Computation*, volume 8402 of *Lecture Notes in Computer Science*, pages 203–215. Springer International Publishing, 2014.

[16] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[17] Christian Liebchen, Marco E. Lübbecke, Rolf H. Möhring, and Sebastian Stiller. The concept of recoverable robustness, linear programming recovery, and railway applications. In *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*, pages 1–27. 2009.

Appendix A. Time results per amount of edges/scenarios on a logarithmic scale

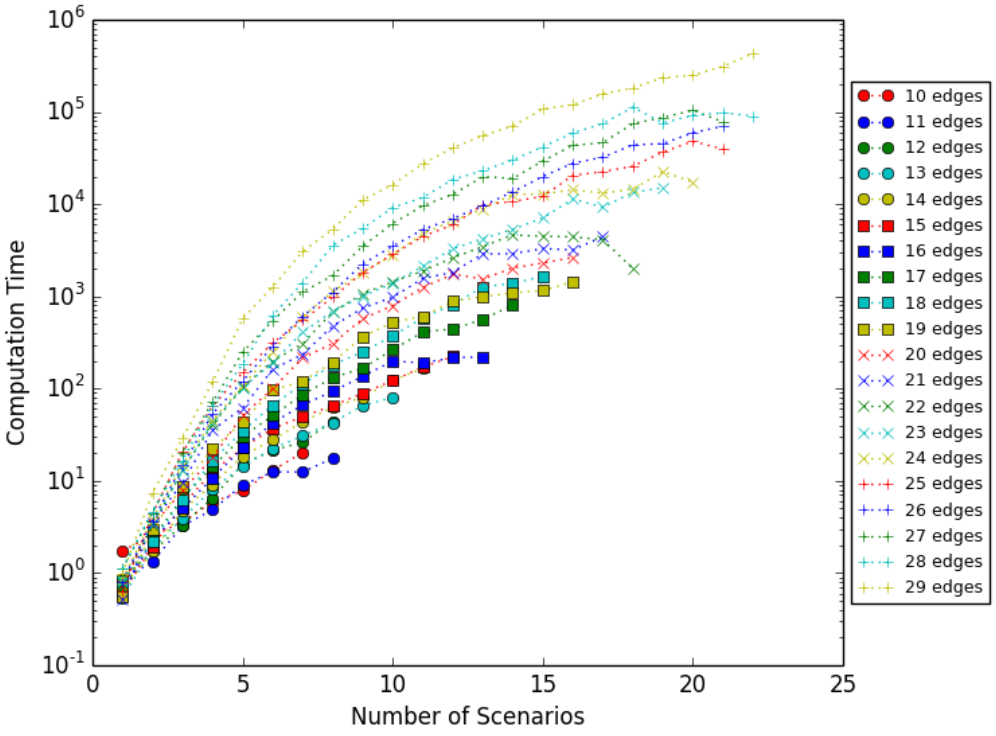

Figure A.6: Time results per amount of edges on a logarithmic scale

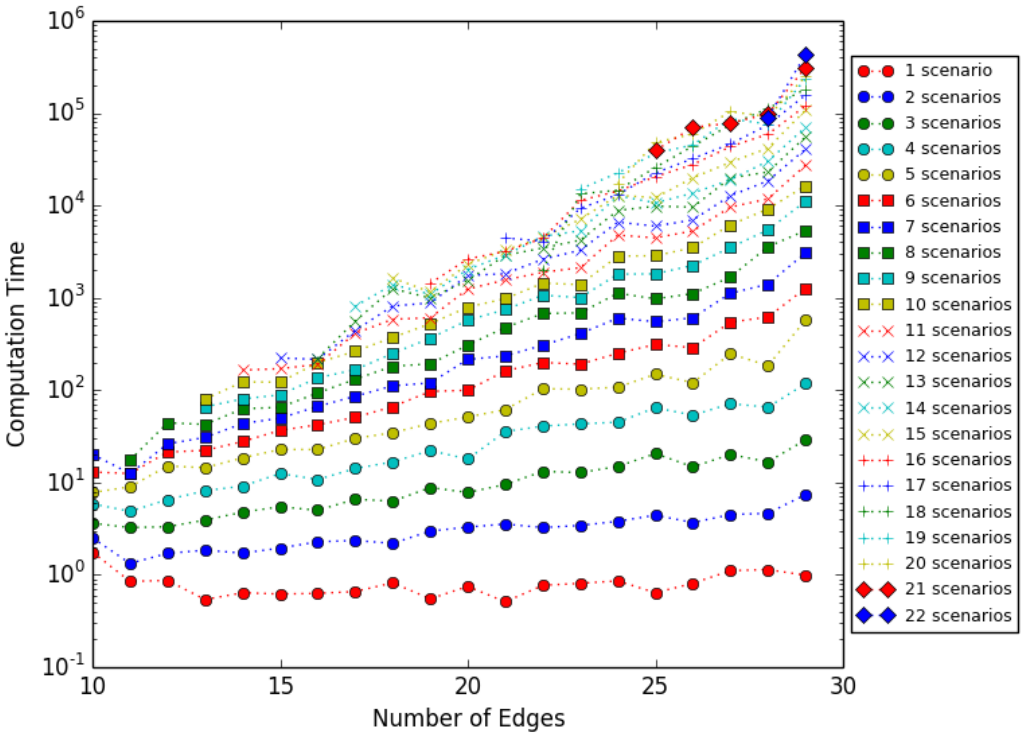

Figure A.7: Time results per amount of scenarios on a logarithmic scale