Master Thesis

A column generation framework for recoverable robustness

Exact and approximation algorithms for knapsack problems with robustness

Author:
Paul Bouman

Supervisors:
Dr. Ir. J.M. van den Akker
Dr. J.A. Hoogeveen

Universiteit Utrecht

ICA-0345369

August 3rd, 2011

A column generation framework for recoverable robustness

Exact and approximation algorithms for knapsack problems with robustness

Paul Bouman

August 3, 2011

When we use combinatorial models for real-life planning problems, we put deterministic data in these models. In reality, this data is often uncertain. Robust optimization tries to take this uncertainty into account. While different approaches to robust optimization exist, this thesis looks at the concept of recoverable robustness. Optimization with Recoverable Robustness tries to find solutions that are robust to some scenarios by applying some means of recovery to an initial solution.

A decomposition framework for reducing these problems into single-scenario problems is presented. The seperate recovery decomposition generates seperate problems for the initial and recovery parts of a solution, while the combined recovery decomposition generates single problems for each scenario that demand an initial solution as well as the recovery solution. We show how we can use column generation to implement these decomposition methods. Example applications to some robustness problems are shown, including robust knapsack problems, a robust weighted independent set problem and the demand robust shortest path problem.

Different variants of the size robust knapsack problem (RKP) are explored and exact algorithms for these problems are presented, as well as upper and lower bound techniques. For one variant the decomposition framework is used to design branch-and-price algorithms. These algorithms, along with several others including local search and dynamic programming techniques, are examined using experimentation on different sets of random instances.

We conclude that the seperate recovery decomposition yields nice results when applied to some instance classes of the size robust knapsack problem. We also conclude that the application of this framework to other problems is an interesting field for future research and experimentation.

Keywords: recoverable robustness, robust knapsack problems, column generation, dynamic programming, demand robust shortest path, Dantzig-Wolfe decomposition

Contents

1. Introduction and Basic Knowledge **1**
1.1. Introduction to Robustness . . . 1
1.2. Linear Programming . . . 4
1.3. Column Generation . . . 4
1.4. Dantzig-Wolfe decomposition . . . 5
1.5. Dynamic Programming . . . 6
1.6. Branch-and-Bound . . . 7
1.7. Local Search . . . 7

2. Models and Decomposition for Robustness Problems **9**
2.1. Robustness Problems . . . 9
2.2. Formal Problem Extension . . . 9
2.3. Framework Overview . . . 11
2.4. Models for Robustness and Disturbances . . . 12
2.5. Separate Recovery Decomposition Model . . . 13
2.6. Combined Recovery Decomposition Model . . . 14

3. The Column Generation Framework **18**
3.1. Scenario Independent General Form . . . 18
3.2. Separate Recovery Decomposition . . . 19
3.3. Combined Recovery Decomposition . . . 21
3.4. Relation to the Dantzig-Wolfe Decomposition . . . 22
3.4.1. Separated Recovery Decomposition . . . 23
3.4.2. Combined Recovery Decomposition . . . 27
3.5. MiniMax Objective Functions . . . 30
3.6. Scenario Generation . . . 32

4. Example Decompositions for Robustness Problems **33**
4.1. A Classroom Problem Example . . . 33
4.1.1. A Robust Knapsack Problem . . . 34
4.2. A Robust Weighted Independent Set Problem . . . 36
4.3. A Minimax Example: Demand Robust Shortest Path . . . 39
4.4. Recoverable Robust Network Flow . . . 41

5. Recoverable Robustness and the Knapsack Problem **44**
5.1. Introduction to Knapsack Problems . . . 44
5.2. Dynamic Programming for KP and related problems . . . 45
5.2.1. Dynamic Programming for KP . . . 45
5.2.2. Balanced Dynamic Programming for SSP . . . 46
5.2.3. Dynamic Programming for PCKP with Trees . . . 49
5.2.4. Dynamic Programming for kKP . . . 52

5.3. Dynamic Programming for Size Robust Knapsack Problems 52
5.3.1. Robustness and the Knapsack Problem 52
5.3.2. Dynamic Programming for F-KP-RC 55
5.3.3. Dynamic Programming for RKP-S 56
5.3.4. Dynamic Programming for RKP-RG 57
5.3.5. Dynamic Programming for RSSP-R and RKP-R 57
5.3.6. Branch and Bound for RKP-R 59
5.4. Upper and Lower-bounds 61
5.4.1. Iterative DP Lower-bound for RKP-R 61
5.4.2. Local Search Lower-bounds 63
5.4.3. LP-Relaxation Upper-bound 65
5.4.4. Lagrangean Relaxation Upper-bound 66
5.4.5. Surrogate Relaxation Upper-bound 67
5.4.6. Recovery Relaxation Upper-bound 69
5.5. Linear Programming Techniques 70
5.5.1. Separate and Combined Phases Decompositions 70
5.5.2. Scenario Column Decomposition 72
5.5.3. Cutting Plane Techniques 73

6. Experiments **75**
6.1. Introduction 75
6.2. Problem instances 75
6.2.1. Generating Items 75
6.2.2. Generating Scenario's 77
6.2.3. Generating Instances 77
6.3. Experiments 78
6.3.1. Choice of Parameters 78
6.3.2. Hardware and Software 78
6.3.3. First Phase Experiments 78
6.3.4. Second Phase Experiments 80
6.3.5. Third Phase Experiments 85

7. Discussion **89**
7.1. Decomposition Framework 89
7.2. Knapsack Problems with Robustness 90
7.3. Experiments 90
7.4. Practical Implications 91
7.5. Future Research 91

8. Conclusion **93**

References **94**

A. Explanation of Techniques Used 97
A.1. Basic Problem 97
A.2. Column Generation 98
A.3. Dantzig-Wolfe Decomposition 101

B. Implementation of Algorithms 103
B.1. Data Structures 103
B.1.1. Items, Scenarios, Problems and the LinkedKnapsack 103
B.1.2. Recovery Knapsack Implementations 103
B.1.3. DPHashTable 104
B.2. Algorithms 104
B.2.1. Separate Recovery Branch and Price 104
B.2.2. Combined Recovery Branch and Price 105
B.2.3. Branch and Bound 106
B.2.4. Exact Dynamic Programming 106
B.2.5. Iterative Dynamic Programming 106
B.2.6. Hillclimbing 107
B.2.7. Simulated Annealing 107
B.2.8. Tabu Search 108

C. Detailed Proofs 109
C.1. Bellman Recurrence 109
C.2. Balanced SSP Algorithm Completeness 110
C.3. Lagrangian Relaxation of RKP-R 110
C.4. A straightforward reduction of 3-Partition to RKP-R fails 113

1. Introduction and Basic Knowledge

1.1. Introduction to Robustness

Suppose we are the management team of a repair facility with repair services on a broad range of products, including kitchen machinery, vehicles, televisions, etc. Since we are the only facility in a considerable area, we have more customers than we can handle. While there are alternatives to in the next state, most customers have a strong preference for our facility, because of the distance. Due to this luxury position, we have devised a unique business model: people can bid on repairs. They tell us what they want to have repaired and how much they want to pay for it, in addition to the repair costs. Two days before the end of each week, we will decide which repairs to accept and which ones to decline, with respect to our production capacity for the next week. Since we want to make a nice living, we want to do this in a way that maximizes our profits. However, if we decline a repair, the customer will go to a facility in the next state.

Solving this decision problem in an optimal way is not trivial. If we have a handful of applications for repairs, we can solve them by hand. But if we have 100 applications each week this will take too much work. Luckily, a lot of research has been done in the fields of Operations Research and Combinatorial Optimization. These fields allow us to create a mathematical model for our problem that can be solved by computers. In fact, our problem is a *Knapsack problem*, a class of problems that has been researched a lot: two full books ([MST90] and more recently [KPP04]) are available.

When we use such a model to solve our problem, we need to make a lot of assumptions when we translate aspects of reality to numbers. Examples of such aspects are the amount of time it takes to execute a certain repair, the availability of certain persons or machines, the time at which a person takes a break, eats his lunch, etc. In many practical cases we don't know the exact values of all those things, so we tend to estimate them. However, in estimation lies the risk of making errors, which can either lead to an infeasible planning or lead to an unnecessary loss in production.

While accepting an unnecessary loss in production is often preferable to accepting a plan that can't be executed, there is a trade-off between the two in many cases. A very unlikely scenario is the situation that a plane crashes down on your facility, halting all production. If you want to take an extremely safe estimate, you would assume that, since it might happen that a plane crashes down, your repair facility can't produce anything at all. Under this assumption the best strategy you can choose is to decline all repairs, leaving your repair facility ineffective to generate any profit at all.

In a general sense, robustness is a property of the solution to a problem that expresses the quality of the solution when applied to certain variations of the problem it was constructed for. Such variations are called scenarios. This implies a solution can be robust for certain scenarios, but not for other scenarios. Since in most cases only an undesirable solution (like *do nothing* or *buy everything*) is feasible for all possible scenarios, it is necessary to select the scenarios that are likely to occur. Now we can quote the definition of robustness given by Stiller [Sti09]: "The fundamental idea of *robustness* is to construct a solution that is *feasible in all* (likely) *scenarios*."

Now one could ask for the definition of *likely*, since this leaves a lot of room for discussion. However, we assume that these likely scenarios are given when we have to find a robust solution to a problem. In other words: the problem of finding the scope of likely scenarios is not a part of the problem we will solve in this thesis (while potentially being an interesting one).

A lot of research was done on ways to create robust solutions to optimization problems. We have the field of Robust Optimization [BTGN09], the field of Stochastic Programming [BL97] and the more recent field of recoverable robust optimization [LLMS07]. One concept is to work with different scenarios, where each scenario contains different assumptions that can possibly occur together. In these cases our solution consists of two part - the initial part contains the decisions that are initially taken and the recovery part contains the decisions that are taken after it is known which scenario is realized.

A major difference between recoverable robust optimization and two-phase stochastic programming lies in the objective. Recoverable robust optimization demands a solution that is feasible for all scenarios when some *restricted* recovery method is allowed, without considering any costs for the recoveries. Two-phase stochastic programming is concerned with finding a solution with a good expected solution value, or a good worst-case value, when we have a single set of first phase choices with imperfect information and multiple sets of second phase choices with all information. In a certain sense, second phase decisions (recourse decisions in the terminology of stochastic programming) are different names for recovery. In recoverable robustness it is also assumed that the method of recovery is limited to simple decisions that can be made by a person. However, two-phase stochastic programming permits you to add constraints on your second phase variables, which will also limit your means of recovery. In short, recoverable robustness deals with the question "What is the best plan that *can be made feasible by simple decisions on the floor* for each scenario?", while two-phase stochastic programming deals with "How can we make a plan that *gives us the best expected or worst-case profit* over each scenario?"

It is not very difficult to think of other real life examples where you have to plan under uncertainty. One could think about a situation where we have to facilitate a group of students with enough classrooms, but we don't know exactly how many people will enroll. While the construction of new buildings is impossible due to budget cuts, it is possible to buy instant classrooms in a container. It is much easier to get a good deal on the containers when you order them early, which implies lower costs for the containers. However, buying a single container at the last moment is probably cheaper than buying three unused classrooms early on. Modeling such a problem is much easier if we can define different scenarios stating how large the group might be. In this case we want our solution to be robust for different sizes of the group of students.

Another example is a case where we want to organize a conference in a field where a lot of speakers are not on speaking terms with each other and won't accept invitations if someone they dislike is invited. Since these speakers are picking up fights all the time, it is probable that in the period between inviting the speakers and the conference itself some additional fights break out. Since we are a respectable conference with a policy to deny any bribery by the tabloid press, we don't want any conflicts to take place at our conference. This means that, in case some new fights break out, we will have to cancel

some speakers. However, the public has a preference for certain speakers and since we get paid by the public, we want to select our speakers in such a fashion that we maximize the expected profit we get from the public, regardless of the speakers we have to cancel. In this case we want our solution to be robust with regard to the fights that can break out between the speakers.

If we take a look at our repair facility example, we might want to take into account the possibility for one of our machines to break down, reducing our production capacity (since our people will need to spend time on the broken machine, instead of the repairs). This gives us two scenarios: one where we have full production capacity for the next week and one where a machine has broken down. Since a customer will go to a repair facility in the next state after you decline his repair, you can't call a customer telling "our machine didn't break down, so we can actually accept your repair which we declined earlier". However, calling a customer telling "our machine broke down, so we can't finish your repair this week" might be more acceptable, especially when you consider this option during your planning process.

We can identify two different periods that are considered during planning. During the initial period we have to make decisions while being uncertain about the scenario that will occur. During the second period we know the occurring scenario and we have to make decisions to turn our initial decisions into a feasible solution taking the recovery methods into account. Since there is a distinction between the period where we have uncertainty and the period where we know what scenario takes place, the recovery decisions are limited by the initial decisions, depending on the method of recovery. This is the concept we refer to as Recoverable Robustness.

In case of the repair facility example, we have to decide which repairs we will accept initially. During recovery we want to decide which customers you need to call in case a machine broke down (if it does that at all). Using this approach we may expect that we are able to accept more repairs than under the safe assumption that we always have a broken machine, but still remain feasible in cases where the machine indeed breaks down.

Of course there are some downsides to this approach. The first one is that the number of possible decisions in our problem rises with the number of scenarios, since we can take different decisions in each of the scenarios.

The second downside is the notion that it becomes more difficult to choose an initial solution that fits all scenarios, depending on the number of restrictions due to recovery. A consequence might be that we can't apply well known efficient algorithms to the problem.

In this thesis we will explore the world between two-phase stochastic programming and recoverable robustness. We will propose a general approach to use decomposition techniques for linear programming with column generation to solve optimization problems with this initial/recovery structure. Special attention goes to new robustness versions of the Knapsack problem, where we will see that problems with certain restrictions can be solved quite efficiently by specialized algorithms, while other restrictions will be solved with the general linear programming technique.

First we will discuss some preliminary knowledge used for different techniques throughout the thesis in Chapter 1. Then we will consider how we can extend problems to robustness variants in Chapter 2. We introduce a general decomposition framework

for robustness problems in Chapter 3 and discuss how it can be applied to the repair facility problem, as well as the conference problem and a shortest path problem where our destination is uncertain in Chapter 4.

A large number of algorithms for knapsack problems related to the repair facility problem are discussed in Chapter 5 and finally some results from experiments based on implementations of some of these algorithms are presented and discussed in Chapter 6.

1.2. Linear Programming

Linear Programming (LP in short) is an optimization technique that searches for a solution to a vector of variables x such that the objective function $z = cx$ is maximized (or minimized) under a system of linear constraints $Ax = b$. This gives the following general form for a Linear Program:

$$\begin{aligned} \max\ (\text{ or} \min\ ) \quad & cx & \\ \text{s.t.} \quad & Ax & = b \\ & x & \geq 0 \end{aligned}$$

If we have integer constraints on some of the variables, we are talking about a Mixed Integer Program (or MIP in short). When all variables have integer constraints, or are binary, we talk about and Integer Linear Program (or ILP in short). If we calculate a solution for the MIP or ILP while ignoring the integer constraints, we talk about the LP-relaxation of the MIP or ILP.

Another popular type of relaxation is the Lagrangian Relaxation. Suppose our objective is to maximize and we have a row i in our constraint system that reads $a^i x \leq b_i$ and a row j that reads $a^j x \geq b_j$. We can relax these constraints by allowing that it may be violated at a cost of λ_i and λ_j per unit. These λ_i and λ_j are called Lagrangean multipliers. We remove the constraints, but we rewrite the objective to $cx + \lambda_i(b_i - a^i x) + \lambda_j(a^j x - b_j)$. Of course, this concept can be done on any number of constraints, which will give us the same number of Lagrangean multipliers. Now we need to choose the right values for these Lagrangean multipliers. The problem of choosing these values is called the Lagrangean dual problem. If our original objective was to maximize, we need to choose our Lagrangean dual multipliers in such a fashion that we minimize the value of the relaxed problem. If we were minimizing, we need to choose the Lagrangean multipliers in such a way that we maximize the value of the relaxed problem. A good textbook on all these concepts is [BJS04] and an overview of early results in Lagrangean Relaxation is [Fis81]. An example and a more detailed explaining can be found in the appendices under section A.1.

1.3. Column Generation

Column Generation is a technique where not all possible variables from the x vector are considered directly when solving the problem. With column generation we start with a restricted set of variables and solve the problem for this restricted set. As a byproduct of

the simplex-method, which is used to solve an LP, we get a dual-vector π, which gives a single value for each constraint (i.e. each row in the constraint matrix). Using this dual vector, we can easily decide if we want to add a new variable x' with a single column a' in the constraint matrix and c' in the objective function to our restricted problem for the set of variables. If we are maximizing, we want to add the variable if $\pi a' < c'$ and if we are minimizing we want to add the variable if $\pi a' > c'$. We call $\pi a' - c'$ the *reduced costs* of the variable (or column) x'. We can also refer to the dual-vector π as the *shadow prices* of our current solution. In general, the problem where we search for an improving variable using the shadow prices is called the *pricing problem*.

Suppose our objective is to maximize. If we solve our pricing problems to optimality and are unable to find any column that improves our current solution after a while, we know we have an optimal solution to the LP-relaxation of the problem. The value of this solution is an upper-bound on the Integer Linear Program and if the current solution is integer, the upper-bound is tight. Now suppose we stop producing columns, while there are still improving columns. In this case we get a lower-bound on the LP-relaxation (which is a lower-bound on the upper-bound). Additionally, any integer solution we are able to find using the generated columns gives a lower-bound on the value of the optimal solution. These observations are related to the Dantzig-Wolfe decomposition [DW60] and the textbook [BJS04] gives an overview of duality theory. A more exact explanation of the technique itself can be found in the appendices under section A.2.

1.4. Dantzig-Wolfe decomposition

Now suppose our constraint system has a specific form such that we can split the vector x into k disjoint variable vectors $x^1, x^2 ... x^k$, b into disjoint constraint vectors $b_0, b_1 ... b_k$ and A into non-zero sub-matrices $D_1 ... D_k$ and $F_1 ... F_k$. Using this approach we can write $Ax = b$ as

$$\begin{array}{ccccccccc} D_1 x^1 & + & D_2 x^2 & + & \dots & + & D_k x^k & = & b_0 \\ F_1 x^1 & & & & & & & = & b_1 \\ & & F_2 x^2 & & & & & = & b_2 \\ & & & & \ddots & & & & \vdots \\ & & & & & & F_k x^k & = & b_k \end{array}$$

The decomposition was introduced by [DW60] and is based on the principle that we can derive an extreme point in $Ax = b$ by writing x as a linear combination of extreme points in each of the $F_k x^k = b_k$ systems. In our restricted master problem we will use variables λ_i^k to represent the ith extreme point x_i^k of the $F_k x^k = b_k$ sub-problem. Note that this λ has nothing to do with the Lagrangean multiplier λ. Using this approach, we can use column generation to create a restricted master problem that combines the extreme points x^k for each k into x. This way, we have k different pricing problems, for each of the $F_k x^k = b_k$ subsystems. This gives us the following restricted master problem:

$$
\begin{array}{llllll}
\text{max or min} & & \sum_k \sum_i (c^k x_i^k) \lambda_i^k & & & \\
\text{s.t.} & & \sum_k \sum_i (D_k x_i^k) \lambda_i^k & = b_0 & & \text{duals: } \pi_0 \\
& & \sum_i \lambda_i^k & = 1 & \forall k & \text{duals: } \pi_k \\
& 0 \leq & \lambda_i^k & \leq 1 & \forall k, \forall i &
\end{array}
$$

Using the duals for a solution to the restricted master problem, we have a pricing problem for each value of k:

$$
\begin{array}{lll}
\text{max or min} & c^k x^k - (\pi_0 D_k) x^k - \pi_k x^k & \\
\text{s.t} & F_k x^k & = b_k
\end{array}
$$

Using the current solution for the restricted master problem, we can find a new column by solving the kth pricing problem and adding the extreme point found for the pricing problem to our restricted master problem. A more thorough explanation of this principle can be found in the appendices under section A.3.

1.5. Dynamic Programming

Another popular way to solve optimization problems is Dynamic Programming, was introduced by Bellman [Bel57]. Dynamic Programming uses a recursive formulation of the optimization problem to divide it into smaller sub-problems. A very simple example is the situation where we want to calculate the nth number from Fibonacci sequence $F(n)$. We can use a recursive formulation to express this problem:

$$
\begin{array}{rcl}
F(0) & = & 1 \\
F(1) & = & 1 \\
F(n) & = & F(n-1) + F(n-2)
\end{array}
$$

If we take a look at the recursive formulation, we can see that finding the nth Fibonacci number can be divided into the sub-problems of finding the $(n-1)$th and $(n-2)$th Fibonacci number. If we use a naive recursive computer program to solve this equation, it will have to do $O(2^n)$ additions. However, we can clearly see that we only need the solution to the sub-problems $F(0) \ldots F(n-1)$. If we use the memory to store these values after we have calculated them, we only need to calculate n values, which takes only $O(n)$ additions, which is much better. Using the memory this way is called *memoization* and is an important part of Dynamic Programming.

If we are looking for an optimal solution and Bellman's Principle of Optimality [Bel57] is applicable to our problem, we can use Dynamic Programming to find an optimal solution to our problem. This principle states that if we are maximizing a value $D(i)$, where i represents the ith choice we are making, the optimality of $D(i)$ only depends on the optimality of values $\max_k D(i-k)$, where the values $i-k$ are found in the recursive definition of $D(i)$. In this case, we call n in $F(n)$ a state variable, since it identifies the current state of the calculation. We may also use formulations that depend on more than just one state variable.

It is possible to create a recurrence for a problem that doesn't adhere to Bellman's optimality principle. You could still use the recursion to find a solution to the problem, but in that case you have no guarantee about the optimality of the solution the algorithm will find.

1.6. Branch-and-Bound

Another common algorithm for optimization is the branch-and-bound method, introduced by [LD60]. This algorithm is based on Depth-First Search, which is described in the excellent textbook [CSRL01]. In addition to the traditional Depth-First Search, branch-and-bound uses upper and lower bounds to eliminate parts of the search tree. Suppose we are maximizing. We start with the depth-first method, until we find a feasible solution. This solution is a lower bound on our solution. Now suppose we have an efficient way to calculate an upper-bound for a certain node in the search tree. Before we expand such a node, we calculate its upper bound and if its upper bound is equal to or lower than the lower bound, we know that expanding this node will not lead to a better solution, if the upper bound is valid.

It is very easy to combine branch and bound with Integer Linear Programming. If the LP-relaxation of an ILP is integer, we know we have the optimal solution. If it has variables with a fractional value, we can branch on a variable by setting its maximum value to the fractional value rounded down and by setting its minimal value to the fractional value rounded up. The value of the LP-relaxation gives a natural upper-bound (in case of maximization). This technique can also be combined with column generation if a good branching rule can be created. With column generation, it is necessary to generate columns for each expanded node in the search tree. Such an algorithm is also called Branch and Price and is described in [BJN+98].

1.7. Local Search

If it is too hard to find an optimal solution to the problem, it is also possible to use techniques to find a good solution, or a solution that will be near optimality. Popular techniques to do this are the local search techniques, the most important ones being Hillclimbing, Simulated Annealing and Tabu-Search. An important aspect of local search algorithms is the neighborhood of a solution. The neighborhood is defined by a number of operations that can be performed on a current solution transforming it into another solution. In case of the repair shop example, adding a certain order that is not scheduled to the current schedule can be seen as an operation. Removing a scheduled order from the current schedule is another operation. The set of solutions that can be reached by performing these operations on a certain solution are called the *neighborhood* of that solution.

Hillclimbing is the simplest of the local search algorithms and is described in [RN03]. It starts with a (possibly random) solution and tries all operations. An operation that improves the value of the solution is chosen. You can vary the way an improving solution is chosen: you can take the best improvement possible with regard to the complete

neighborhood (which is an usual way for Hillclimbing), but you can also take the first improvement you find (which is used during the experiments in Chapter 6). This is repeated until no operation can be performed that will improve the current solution and terminates. Hillclimbing walks directly to a local optimum. It is possible to run Hillclimbing a number of times on different random starting solution, keeping the best solution found.

Simulated Annealing was first described in [KGV83] and performs a random operation and checks whether the value of the solution has improved. If it has, the operation is accepted and the new solution becomes the current one. If it hasn't improved, the degradation of the value is weighed against the current temperature and a random factor to decide whether the degradation should be accepted or declined. Over time, the temperature will cool down in such a fashion that degradations are accepted less often.

Tabu-Search was introduced by [GL97] and checks all possible operations in each step and takes the best possible neighbor (which may have a worse value) that is not the the Tabu-list. The Tabu-list is a list of a certain constant length k that contains the k last visited solutions (or properties of the k last visited solution). When we find a new solution, we add it to the front of the tabu-list and remove the last element from the list. If k has a large value, it is possible that all neighbors of the current solution are in the Tabu-list. In such a case the algorithm terminates.

When using Local Search, it is important to choose a neighborhood that is appropriate for the algorithm used. In case of Simulated Annealing and Tabu-Search, it is also important to choose the values of the parameters in an appropriate fashion. More information about algorithms for searching is given in the textbook [RN03].

2. Models and Decomposition for Robustness Problems

2.1. Robustness Problems

When we consider some optimization problem, for example a Knapsack Problem, or a Shortest Path problem, we have some question and we must decide on a (possibly best) solution. If we consider the Shortest Path problem, our question can be written as 'Considering this graph G, what is the shortest path from node v to node w'. Normally, we assume that in a single problem instance G, v and w are fixed. When we are dealing with uncertainty, an example may be that we are unsure about which node will become node w. The algorithm which calculates the shortest path, calculates all shortest paths from a single source. If we are only uncertain about which node will become w, we can run the shortest path algorithm, wait until we know the exact node w and use the solution that was already calculated using the algorithm.

Waiting for information to become available can introduce higher costs for some decisions. For example, if we would have to buy plane or train tickets, there is a big chance they are cheaper if we buy them early on. Even if we assume we know the exact price in the future, simply running the shortest path algorithm to calculate the shortest path to all nodes won't work. Another observation is the fact that if the edges in the graph represent airline or train tickets, it is probable that you can't sell them after they are bought. This implies there is some constraint on the solution based on the moment at which decisions are made. Making a decision at a certain moment will influence the possible decisions at a later time.

We can make two important observations: in modeling robustness problems we are confronted with decisions whose effects are uncertain and the problem model that takes robustness into account can be seen as an extension of our regular problem models. When we have an algorithm for such a regular problem model, this gives a lot of information and insight into how we can solve such a model. Therefore, it would be preferable to make use of this information when we extend such a model with the notion of robustness.

Suppose we are asked to solve a robustness problem by hand, but are given access to an algorithm for the non-robustness variant. We could try to work on a what-if basis: assume a single scenario happens and try to use the given algorithm on a problem instance that represents this single scenario. If we repeat this for each scenario, we can try to construct a solution to the robustness problem by combining the separate non-robust solutions.

In this chapter we will discuss how we can use decomposition techniques to solve these problems using such approach. In Sections 2.2 and 2.3 we will discuss the extension of problems to take robustness into account. Then we will discuss the overview of the framework and the existing models for robustness. In Sections 2.5 and 2.6 we will discuss the computational methods from our framework.

2.2. Formal Problem Extension

When we work with robustness, we have two types of decisions: decisions that should be taken now, which we will call initial decisions, and decisions that can be taken at a

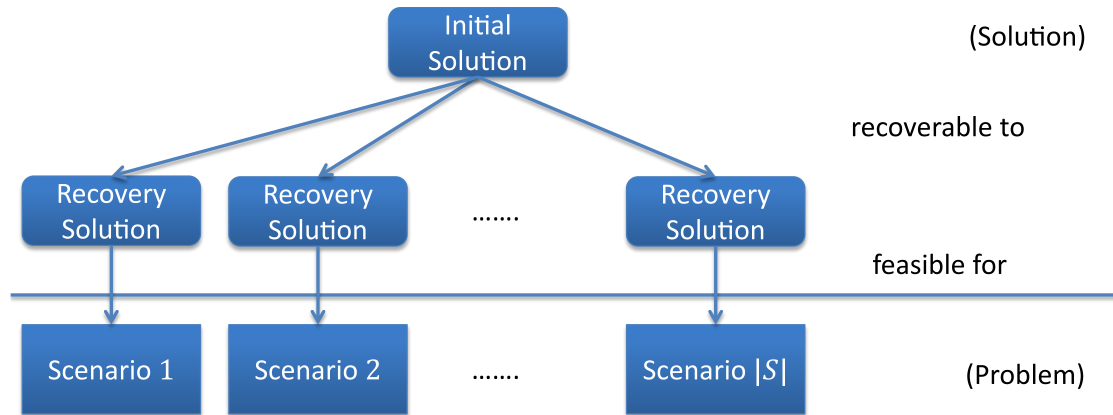


Figure 1: Representation of the Solution Structure of a Robust Problem

later time, which we will call recovery decisions. We are also confronted with different scenarios, for which recovery may or may not be necessary given some initial solution. We will introduce some notation:

- **Scenarios** We have a set S that contains possible scenarios, represented by indices $1 \ldots |S|$.
- **Initial Variables** The vector x contains decision variables that represent initial decisions.
- **Recovery Variables** A vector y^s contains decisions variables that represent recovery decisions for a single scenario $s \in S$.
- **Initial Costs** The initial costs can be represented using either a cost function $f(x)$ or a vector c^1 in case of linear costs or profits in relation to the decision variables in x.
- **Recovery Costs** The recovery costs for a scenario s can be represented using either a cost function $f(y^s, s)$ or a vector c_s^2 in case of linear costs or profits in relation to the recovery variables.

Besides having a distinction in types of variables, our constraints will also have a special structure: we can have constraints on the initial decision variables (initial constraints), on the recovery decision variables (recovery variable constraints) and constraints that state what recovery options are possible given a certain initial solution, for a single scenario (the recovery constraints). A representation of this structure and its relation to the problem is presented in Figure 1.

Now let us consider the repair facility example. Our decision variables in x will model the choices to accept or decline certain repairs initially. We can use the recovery decision variables in the vector y^s for a certain scenario s in two ways: they can model the choices

to accept or decline certain repairs after the capacity becomes known or we can use them to model the choice to reject a repair we accepted earlier. We will use the first option, because this gives a nicer constraint.

The constraints in this example are the capacity constraint which we model using both an initial constraint (for the initial capacity) and the recovery variable constraints (for the recovery capacities). The constraint to decline repairs that have been accepted must be modeled using recovery constraints, since it relates the initial decision variables to the recovery variables. We will discuss this problem more thoroughly in Section 4.1.1.

2.3. Framework Overview

It is not difficult to see that if we have a pool of potential solutions for the y^s vectors for each $s \in S$ and also a pool of potential solution for the x vectors, we can create a solution for the problem by selecting a y^s vector from each pool and a x vector from the pool, in such a fashion that each selection for y^s can be recovered from the selection for x. If we have a simple method to see if a certain y^s vector is recoverable from a certain x vector, we can look at each x vector in the pool, and select the best matching y^s vector for each scenario s. This way, we can check for each vector if it can be made feasible for each scenario at all and calculate the value of the best combination for each x vector. Now let us assume that we can express these restrictions on the recovery by simple means on these vectors. Since we can now check whether an initial and recovery solution are compatible, we remain with a single problem: filling the pools with potentially interesting solutions.

Our intuition tells us that it will be a lot easier to adapt an algorithm for the original problem to be suitable for finding a single x or y^s vector, than for finding the complete solution, since finding a shortest path to a single sink or selecting repairs for a single capacity gives less to worry about. Given some combination of vectors, we are interested in finding vectors that improve the current situation. If we modify the costs or profits of the variables in the vectors, we get different problems that may result in different solutions. We must use some method to modify the objective of these pricing problems.

But what if we don't want to express the compatibility of an initial and a recovery solution by simple means? Suppose we have a pool for each scenario s, such that the pool contains tuples (x, y^s), which are combinations of a solution vector x and a solution vector y^s, such that y^s is recoverable from x. If we select a tuple from each pool, such that for each tuple the x part is equal to all the x parts of the other tuples we selected, we have a valid solution to our problem. Now if we can adopt our original algorithm in such a way that we can find interesting combinations of a x and a y^s, we can use column generation to find a solution to our problem. This time, the algorithm will give bonuses and maluses on the variables in the x vector, that should be taken into account by the algorithm.

We will refer to the first approach as the separate recovery decomposition, because we need to find separate initial solution vectors x and vectors y^s for the recovery solution of each scenario s. We will refer to the second approach as the combined recovery decomposition, since we need to find combinations of a x and a y^s vector for each of the

scenarios s. A schematic overview of the process of both decompositions is discussed in Sections 2.5 and 2.6.

2.4. Models for Robustness and Disturbances

Now let us consider a general minimization problem

$$P_{\min} = \min\{f(x)|x \in X\}$$

where x is a vector of decision variables and X is the set of feasible solutions (or feasible region for the vector x).

When we consider robustness, we face the situation where we are uncertain about the set X to a certain degree. This introduces the difficulty that we cannot simply take any $x \in X$ with a minimal value for $f(x)$. The field of *robust optimization* ([BTGN09], [BS04]) presents models for finding solutions that are feasible for likely disruptions in X. However, since $f(x)$ is unchanged, this approach may come with great costs regarding the value of the objective.

The field of *stochastic programming* [BL97] presents models that consider the expected value with regard to disturbances as an objective. The disturbances are modeled as random vectors ξ, based on given probability distributions. In case of two-phase (or multi-stage) stochastic programming, recourse actions are introduced, that may be taken with respect to each realized vector ξ, a recourse matrix (which imposes limitations to the recourse actions) and a technology matrix (which imposes limitations to between the recourse actions and the solution).

More recently, the concept of *recoverable robustness* was introduced [LLMS07]. Recoverable robustness introduces a model where a basic problem, like $P_{\min}$ is extended with a set of scenarios S and a set of admissible recovery algorithms $\mathcal{A}$. The result of recovery algorithm $A(x, s) \in \mathcal{A}$ given a solution x and a scenario s, is a feasible solution for scenario s that is recoverable from x. Now, a *Recoverable Robust Minimization Problem* ($RRP_{\min}$) is defined as

$$RRP_{\min} = \min\{f(x) + \sum_{s \in S} g(y^s, s)|x \in X, A \in \mathcal{A}, \forall s \in S : A(x, s) = y^s\}$$

The relation between recoverable robustness and the other approaches is discussed in [LLMS07]. To summarize, strict robust optimization has the disadvantage of "finding unacceptably expensive or conservative solutions"[LLMS07]. There are some major similarities between two-phase stochastic programming and recoverable robustness, but stochastic programming puts an emphasis on the use of probability distributions for the uncertainty and using expected costs for the optimization objective. In addition to this, there is a difference in the use of a set of admissible recovery algorithms, and the use of a technology and a recourse matrix.

For our models, we will consider a modification of the RRP. First we will restrict our problem to a single admissible recovery algorithm $A \in \mathcal{A}$. This is acceptable, since we may solve our resulting problem for each $A \in \mathcal{A}$ and just take the best solution we find

that way. In addition to this, we define two sets of feasible solutions (or feasible regions) for each scenario $s \in S$. We define feasible scenario solution sets

$$\forall s \in S : Y^s \supseteq \{y^s | \exists x : A(x,s) = y^s\}$$

While it is possible to use $=$ instead of $\supseteq$, our approach will still work if Y^s contains solutions that are not reachable trough recovery. Since these solutions are useless for our final solution, we should try to avoid including these in our Y^s sets as much as possible. We define feasible recovery sets

$$\forall s \in S : R_s = \{(x, y^s) | A(x,s) = y^s\}$$

Using these definitions, we reformulate the $RRP_{\min}$ to a *Feasible Region Recoverable Robust Minimization Problem* ($FRRRP_{\min}$). This problem is defined as

$$FRRRP_{\min} = \min\{f(x) + \sum_{s \in S} g(y^s, s) | x \in X, \forall s \in S : y^s \in Y^s, \forall s \in S : (x, y^s) \in R_s\}$$

2.5. Separate Recovery Decomposition Model

Now suppose that $FRRRP$ is difficult to solve. When we enumerate the full set X and each set Y^s and take the best possible combination that is in R_s, we get the best solution. However, the number of combinations is usually very large, being $|X| \prod_{s \in S} |Y^s|$. In addition to this, we will ignore most of the solutions in these sets. The idea is to consider only limited subsets for X and each Y^s, that approximate the relevant parts of these sets with regard to the $FRRRP$. We therefore introduce a set

$$X' \subseteq X$$

and sets

$$\forall s \in S : Y'^s \subseteq Y^s$$

We will now start with small sets, expanding these sets in an incremental way. We propose the *Separate recovery decomposition model*, where we define a *Separate Master Problem SMP* for finding a good combination based on our restricted sets, a *Separate Initial Pricing Problem $SIPP$* for expanding the set X' and *Separate Recovery Pricing Problems $SRPP^s$* for expanding the sets Y'^s .

$$SMP_{\min} = \min\{f(x) + \sum_{s \in S} g(y^s, s) | x \in X, \forall s \in S : y^s \in Y'^s, \forall s \in S : (x, y^s) \in R_s\}$$

Note how the SMP still contains the original set R_s - while the x and each y^s are chosen from the restricted sets, the SMP is still concerned with recovery. Since we want to expand our restricted sets to let the SMP find better solutions, we will introduce a function π and a function π_s for each $s \in S$. The purpose of these functions is to

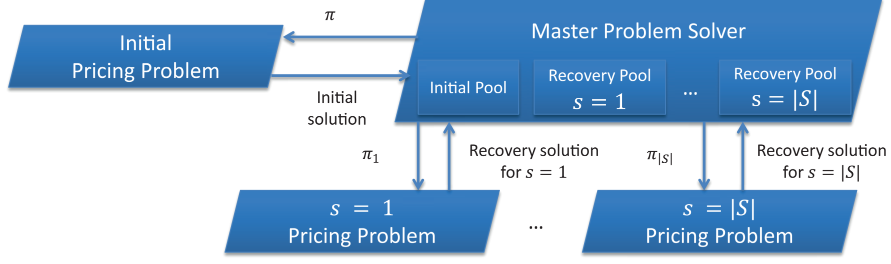


Figure 2: Separate Recovery Decomposition Process Overview

pass information from the SMP to our pricing problems, in such a way that improving solutions are found. Using these functions, we introduce the *Separate Recovery Initial Pricing Problem* $SIPP$ and a *Separate Recovery Pricing Problem* $SRPP^s$ for each scenario $s \in S$.

$$SIPP_{\min} = \min\{f(x) - \pi(x) | x \in X\}$$

$$SRPP^s_{\min} = \min\{g(y^s, s) - \pi_s(y^s) | y^s \in Y^s\}$$

We must note that the quality of the final solution to the SMP depends mostly on the π and π_s functions - if the optimal solutions are not found by the pricing problems, the SMP just gives an approximation. The full process of the Separate recovery decomposition is given in Algorithm 1, while Figure 2 gives an overview of this process. The major advantage of this approach is that the pricing problems do have the nice property that they are only defined in terms of a single feasible set X or Y^s. If we have algorithms available to find a solution to such a single, feasible set of solutions, we can embed these algorithms in the process of finding solutions to a recoverable robustness problem.

2.6. Combined Recovery Decomposition Model

Let us consider a different way to decompose the $FRRRP$ into sub-problems. The Separate Combined Recovery Decomposition has a master problem that has to take the recovery into account. We can create a different model, the *Combined Recovery Decomposition Model* that moves the recovery into the sub-problems as well. For this purpose, we introduce a *Combined Master Problem* CMP and a *Combined Pricing Problem* CPP^s for each $s \in S$. The $FRRRP$ gives us a feasible recovery set R_s for each scenario $s \in S$. In the CMP, we will consider restricted recovery sets $R'_s \subseteq R_s$ for each $s \in S$.

$$CMP_{\min} = \min\{f(x) + \sum_{s \in S} g(y^s, s) | \forall s \in S : (x'_s, y^s) \in R'_s, \forall s \in S : x = x'_s\}$$

This time, we have a single pricing problem for each scenario. We define a function π^s for each scenario, to provide the pricing problem with information from the current

Algorithm 1 Scheme for the Separate Recovery Decomposition

x_0 is some initial solution
Create a set X'
Add x_0 to X'
Initialize Problem $SIPP$ using X'
for $s \in S$ **do**
  Create a set Y'^s
  Calculate $y_0^s \leftarrow A(x_0, s)$
  Add y_0^s to Y'^s
  Initialize Problem $SRPP$ using Y'^s
end for
Initialize Problem SMP using X' and each Y'^s
while improvement **do**
  Solve SMP
  Calculate π
  $x' \leftarrow$ SOLVE $SIPP$ using π
  if x' improves SMP **then**
    Expand X' to $X' \cup \{x'\}$
  end if
  for $s \in S$ **do**
    Calculate π_s
    $y'^s \leftarrow$ SOLVE $SRPP^s$ using π_s
    if y'^s improves SMP **then**
      Expand Y'^s to $Y'^s \cup \{y'^s\}$
    end if
  end for
end while
return SOLVE SMP

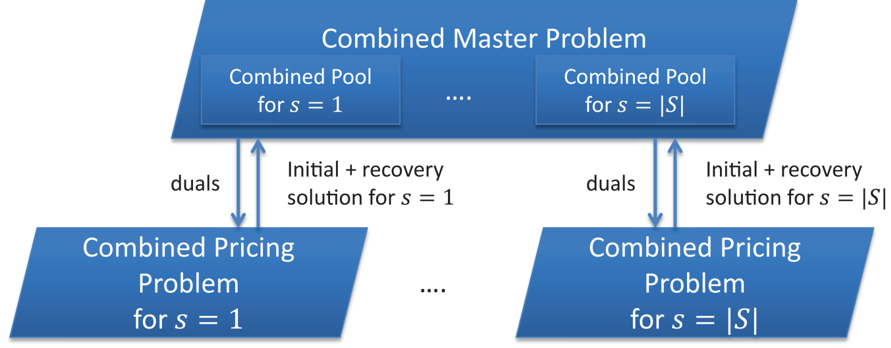


Figure 3: Combined Recovery Decomposition Process Overview

state of the CMP. These functions should be chosen in such a fashion that the resulting solution for the CPP^s can improve the solution space of R'_s and thus of CMP.

$$CPP^s_{\min} = \min\{g(y^s, s) - \pi^s(x) | x \in X, y^s \in Y^s, (x, y^s) \in R_s\}$$

Algorithm 2 Scheme for the Combined Recovery Decomposition

x_0 is some initial solution
for $s \in S$ **do**
  Create a set R'_s
  Calculate $y^s_0 \leftarrow A(x_0, s)$
  Add (x_0, y^s_0) to R'_s
  Initialize Problem CPP^s using R'_s
end for
Initialize Problem CMP using the generated sets
while improvement **do**
  SOLVE CMP
  for $s \in S$ **do**
    Calculate π^s
    $(x', y'^s) \leftarrow$ SOLVE CPP^s using π^s
    if (x', y'^s) improves CMP **then**
      Expand R'_s to $R'_s \cup \{(x', y'^s)\}$
    end if
  end for
end while
return SOLVE CMP

Again, the quality of this method depends mostly on the quality of the π^s functions. The nice property of this decomposition is the fact that a single pricing problem CPP^s contains only the three feasible spaces X, Y^s and R_s. If we can think of an algorithm that finds both an initial and a recovery solution for a single scenario, we can use such

an algorithm for the pricing problem. The process is formally presented in Algorithm 2 and an overview is given in Figure 3.

We could even consider the $CPP^s_{\min}$ in terms of the original problem $RRP_{\min}$. Suppose we have some π^s. For each $s \in S$, let us find the solution to $\min\{-\pi^s(x)|x \in X\}$ and let us apply the recovery algorithm to find $y^s = A(x,s)$. We can add (x, y^s) to R'_s. While quality depends on what π^s is used, this approach can give an approximation to the solution to $CMP_{\min}$.

3. The Column Generation Framework

3.1. Scenario Independent General Form

In this chapter we will consider how we can implement the framework using column generation. To achieve this, it is necessary to take a look at linear programming techniques. In this chapter we will formulate linear program for the master problems and show how they can be used to implement the framework. We also show we can derive these models by application of the Dantzig-Wolfe decomposition to a general linear program for Recoverable Robustness Problems. In this section, we we will describe some properties of robustness problems.

Let us consider the structure where we have a x variable vector for the initial solution and multiple y^s variable vectors for the recovery solution of each scenario $s \in S$. The structure of the problem will follow a tree structure. Based on these variable vectors and the presented structure, we will formulate some required properties of our problem.

- **Scenario Independence (SI)** The possible values of a single y^s vector do only depend on the values of the x vector, not on other $y^{s'}$ vectors such that $s' \neq s$.

- **Feasibility Algorithm (FA)** This algorithm needs to find a feasible solution, or be able to report that the problem is infeasible. This is to generate initial solutions to initialize the LP models. An example for the KP is to return an empty knapsack.

- **Linear Adaptable (Initial and/or Recovery) Objectives (LAO)** The objective associated with a variable vector x or y^s can be adapted by adding or subtracting some value π_i for each variable x_i with regard to the original objective factor c_i^1. It is important that the algorithm we use to solve our sub-problems can cope with such adaptations to the problem.

- **Recovery Algorithm (RA)** If there is a Recovery Algorithm, we have some way to calculate the best solution vector y^s for a given scenario s and a fixed vector x. Recoverable Robustness implies such an algorithm exists. We can apply this to each initial solution we have found so far to get a good solution to the full problem.

- **Initial Solution Partitioning Method (IPM)** To use branching in combination with Column Generation, we need to be able to partition the solution space of the initial solution into disjoint partitions. It is important that the other properties still hold in each of these partitions. A possible way to do this, is by restricting the domains of the variables. Suppose we can take a variable x_i that currently has a domain in the range $\{l \dots u\}$ and change its domain to either $\{l + k \dots u\}$ or $\{l \dots u - k\}$ for some $u - l \leq k > 0$. We call this property Restrictable Variable Domains (RVD) and IPM is implied by this property.

- **Linear Recovery Constraints (LRC)** A problem has linear recovery constraints, if we can write the limitations between a certain recovery scenario vector y^s and the initial scenario vector x using two matrices A_1, A_2 and some vector of constants b

	Exact Solution	Estimate Solution
Both Decompositions	SI IPM FA	SI RA FA
Separate Recovery Decomposition	LAO LRC	LAO LRC
Combined Recovery Decomposition	LAO1	LAO1

Table 1: Required properties for both decompositions and answer types

using either $A_1x + A_2y^s = b$, $A_1x + A_2y^s \leq b$ or $A_1x + A_2y^s \geq b$ for each possible scenario $s \in S$.

The most important property is Scenario Independence. SI is very plausible, since it is closely related to the tree-like structure of the robustness extension. The LAO property can apply to only the initial variables (LAO1) or both (LAO). The Linear Recovery Constraints are necessary in case of the separate recovery decomposition.

We can use the decompositions to calculate an exact solution, or to calculate an estimate solution. The exact solution will use Branch-and-Price to find the exact solution to the problem, while the Estimate Solution will give a lower and upper-bound on the solution value, as well as an estimate solution itself.

In Sections 3.2 and 3.3 we will discuss a situation where we express the master problems with a linear programming model to derive dual values for the pricing problems. In Section 3.4 we will show how to derive all linear programs involved using the Dantzig-Wolfe decomposition.

3.2. Separate Recovery Decomposition

Now let us look at the separate recovery decomposition. We will assume Scenario Independence, Linear Adaptable Objectives, Linear Recovery Constraints and a Feasibility Algorithm. We have a variable vector x and variable vectors y^s. Let us define a single pool X', that contains fixed vectors with values for the x variables. Also, let us define a pool Y'^s for each scenario s, that contains fixed vectors with values for the y^s vectors. Each pool will contain extreme points with indices k in pool X' and indices j in a pool Y'^s for each scenario $s \in S$.

We can represent a full solution by selecting a single vector from the X' pool and a single vector for each Y'_s pool, if we assume the selected vectors satisfy our recovery constraint. Now let us consider binary variables u_k, where each index k represents a vector from pool X'. Let us also consider binary variables v_j^s for each scenario $s \in S$, where index j represents a vector from pool Y'^s.

k	:	an index over the set of feasible initial solutions X
j	:	an index over the set of feasible recovery solutions Y^s for scenario s
x_k	:	the kth initial solution in X
y_j^s	:	the jth recovery solution in Y^s for scenario s

$$u_k = \begin{cases} 1 & \text{the } k\text{th initial solution (i.e. } x_k\text{) is selected} \\ 0 & \text{otherwise} \end{cases}$$

$$v_j^s = \begin{cases} 1 & \text{the } j\text{th recovery solution (i.e. } y_j^s\text{) is selected for scenario } s \\ 0 & \text{otherwise} \end{cases}$$

Now let us assume that each feasible recovery set $R_s = \{(x_k, y_j^s) | A(x_k, s) = y_j^s\}$ is bounded by a polyhedron, which implies that the limitation on recovery can be expressed using linear constraints (this is the LRP-property from Section 3.1). Let us express these feasible recovery sets using a linear system of equations. Now, for each $s \in S$, let us introduce matrices A_1^s, A_2^s, vectors b_1^s such that y_j^s is recoverable from x_k if and only if

$$A_1^s x_p + A_2^s y_j^s \leq b_1^s$$

$$\begin{array}{llllll}
LSMP_{\min} = & \min & \sum_k (c^1 x_k) u_k + \sum_{s \in S} \sum_j (c_s^2 y_j^s) v_j^s & & & \\
& \text{s.t.} & \sum_k (A_1^s x_k) u_k + \sum_j (A_2^s y_j^s) v_j^s & \leq b_1^s & \forall s \in S & |S| \text{ dual vectors } \pi_s \\
& & \sum_k u_k & = 1 & & \text{dual } \pi^X \\
& & \sum_j v_j^s & = 1 & \forall s \in S & |S| \text{ duals } \pi^{Y^s} \\
& & u_k \in \{0,1\} & & \forall k & \\
& & v_j^s \in \{0,1\} & & \forall s \in S, \forall j &
\end{array}$$

Our main goal is to find new columns for the X' or Y'^s pools, that improve the current solution. We can use the duals given by the current solutions to adapt the objectives of our pricing problems. Since we assumed linear adaptable objectives, we can use the duals to derive linear factors for the variables. If we take $\sum_{s \in S} (\pi_s A_1^s)$, we get a factor for each variable in x. Also, if we take $\pi_s A_2^s$, we get a factor for each variable in y^s. Using these factors and the LAO assumption, we can use the original algorithm (that will take the original constraints into account) to find an interesting new column to improve our current complete solution, or to detect that we cannot improve our current solution further.

If we cannot improve the current solution, we might arrive at a situation where the current solution is fractional. To calculate an estimate, we can assume we have a Recovery Algorithm and run it on all solutions from our X' pool and choose the best one as the solution. In that case we can use the value of the fractional solution to determine the gap between the lower and the upper-bound of the derived solution. If we want to calculate

an exact solution and take a look at a variables that have different values in the selected columns from pool X' (multiple columns must be selected, since the solution is fractional). If we have RVD, we can branch on a variable that has a different value in the selected columns by applying the assumption of restrictable variable domains. Suppose some variable x_i has value ω, we restrict the domain of x_i by putting its upper bound to $\lfloor \omega \rfloor$ for a new branch and putting its lower bound to $\lceil \omega \rceil$ for another new branch. This way, we will forbid columns that have a value not inside the new domain of x_i. By repeating this process in the branches, all variables in x will be fixed eventually.

By branching based on a fractional solution, we know there must always be at least two vectors in the X' pool that are selected fractionally. Since these vectors must differ in some variable x_i inside these vectors, the domain of this variable x_i has enough freedom to select either one of the vectors. Clearly, we can restrict the domain of this x_i in such a fashion that we forbid one of the vectors, while keeping the other vector feasible. The advantage is that we always keep a feasible vector in the X pool. In case there is some Y'^s pool that has no vectors that are recoverable from any vector in the X' pool, we can use a Feasibility Algorithm to create feasible vectors. However, since the variables in vectors of the Y^s pools are not restricted at all, doing this is only necessary if we all remaining vectors in the $'X$ are not recoverable for certain scenarios. When this is known not to be the case, a Recovery Algorithm can be used on the remaining solutions in the X' pool, instead of the Feasibility Algorithm. In our experience so far this aspect of branching was not difficult to solve. However, one should keep this in mind when using the technique. An obvious option is to see if one can add some fallback option to each of the Y'^s pools, which has a bad objective value, but is recoverable from each conceivable vector in the X' pool.

3.3. Combined Recovery Decomposition

While the Separate Recovery Decomposition tries to combine initial and recovery solutions in such a fashion that they satisfy the recovery constraint, the Combined Recovery Decomposition takes another approach: we try to combine combinations of initial and recovery solutions in such a fashion that all initial solutions are equal.

The idea is that selecting the same initial solution for each scenario gives the same solution as selecting a single initial solution and a recoverable solution for each scenario. When we assume that the combinations of a initial and a recovery solution under our consideration satisfy the recovery constraints, our main concern is to look for a combination for each scenario with the only constraint that the initial solution of all these combinations are equal.

p	:	an index over the set of feasible initial solutions X
q	:	an index over the set of feasible recovery solutions Y^s for scenario s
x_p	:	the pth initial solution in X
y_q^s	:	the qth recovery solution in Y^s for scenario s
R_s	:	the set containing all pairs $(x_p, y_q^s) \in X \times Y^s$ for scenario $s \in S$ such that y_q^s is recoverable from x_p

$$w_{pq}^s = \begin{cases} 1 & \text{for scenario } s \quad \text{the } p\text{th initial solution (i.e. } x_p) \text{ is selected,} \\ & \qquad\qquad\qquad \text{the } q\text{th recovery solution (i.e. } y_q^s) \text{ is selected,} \\ & \qquad\qquad\qquad (x_p, y_q^s) \in R_s \\ 0 & \text{otherwise} \end{cases}$$

Now let us write the Combined Master Problem CMP as an ILP to get the Linear Combined Master Problem LCMP:

$$\begin{array}{llllll} LCMP_{\min} = & \min & c^1 x + \sum_{s \in S} \sum_{(p,q)} (c_s^2 y_p^s) w_{pq}^s & & & \\ & \text{s.t.} & \sum_{(p,q)} w_{pq}^s & = 1 & \forall s \in S & |S| \text{ duals } \pi^s \\ & & \sum_{(p,q)} x_p w_{pq}^s - x & = 0 & \forall s \in S & |S| \text{ dual vectors } \pi'^s \\ & & w_{pq}^s \in \{0, 1\} & & \forall s \in S, \forall (p,q) & \end{array}$$

When we solve the pricing problems based on the duals we will eventually arrive at a point where we cannot improve the current solution, we can use the pools to find an estimate solution. This can be done either by grouping the vectors in the pools by their initial solution parts and taking the best one, or by deriving a initial solution from our x_i variables and apply the recovery algorithm to find a full solution. If we want an exact solution branching is necessary. This cannot be done on the x_i variables in the master problem, so it is necessary to branch on the solution space (ILM), for example by branching on variables in the vector x in the pricing problems. Of course, partitioning implies that we cannot select some vectors from the currents pools during the time we remain in this branch. In some situations this may lead to a feasibility problem, which can be solved by using a Feasibility Algorithm. If the Feasibility Algorithm yields no feasible solution, we know the current branch is infeasible and we may stop exploring it.

3.4. Relation to the Dantzig-Wolfe Decomposition

We will now explore the relation between the two decompositions and the Dantzig-Wolfe decomposition for Linear Programs. We will begin with a general form for linear programs that represent the recoverable robustness problems in an ILP-form. We will use a variable vector x to describe our initial decision variables and a variable vector y^s for each $s \in S$ for the recovery decision variables for a scenario s.

$$
\begin{array}{llllll}
LRRP_{\min} & = & \min & c^1x + \sum_{s\in S} c_s^2 y^s & & & \text{objective} \\
& & \text{s.t.} & A_1^s x + A_2^s y^s & \leq b_1^s & \forall s \in S & \text{recovery constraints} \\
& & & A_3 x & \leq b_2 & & \text{initial variable constraints} \\
& & & A_4^s y^s & \leq b_3^s & \forall s \in S & \text{scenario constraints}
\end{array}
$$

We refer to this form as the LRRP. This general form removes the possibility to express constraints that are not *scenario independent*, while allowing other forms of direct dependence. Initial decisions go into the x vector, recovery decisions go into the y^s vectors. The constraints on the initial decisions states which initial solutions are feasible and can be expressed using A_3 and b^2. The constraints on the recovery decisions for a scenario s can be expressed using A_4^s and b_s^3. To express our recovery constraints we can use the respective A_1^s and A_2^s matrices and the b_1^s vector, for each scenario. Since we have split our variables into different vectors, the costs of the variables will also be split into separate vectors: c^1 contains the costs of the initial decisions and each c_s^2 vector expresses the costs for the recovery decisions for a certain scenario s.

3.4.1. Separated Recovery Decomposition

Let us take the LRRP from the last section. We will decompose it by separating the initial and recovery vectors. The idea of the separated recovery decomposition is to find initial and recovery solutions separately, by moving the $A_3x = b_2$ and $A_4^s y^s = b_3^s$ constraints into separate pricing problems, retaining the link constraints $A_1^s x + A_2^s y^s = b_1^s$ in the master problem. In fact, this is a rather straightforward application of the Dantzig-Wolfe decomposition, where the separate $A_3x = b_2$ and $A_4^s y^s = b_3^s$ systems become the sub-problems. The nice part about this is that it decomposes the problem into a single sub-problem for the x vector and a single sub-problem for each separate y^s vector.

$$
\begin{array}{llll}
\min & c^1x + \sum_{s\in S} c_s^2 y^s & & \\
\text{s.t.} & A_1^s x + A_2^s y^s & \leq b_1^s & \forall s \in S \\
& A_3 x & \leq b_2 & \\
& A_4^s y^s & \leq b_3^s &
\end{array}
$$

To apply the Dantzig-Wolfe decomposition on this model, we need the representation theorem. To simplify things, let us assume that the our solution space is bounded, so may consider a simplified version of the representation theorem.

Theorem 3.1 (Representation Theorem for Bounded Sets, based on [BJS04])**.** *Let $X = \{x|Ax \leq b, x \geq 0\}$ be a nonempty (polyhedral) bounded set. Then set of extreme points $\{\hat{x}_1, \ldots, \hat{x}_l\}$ associated with X is nonempty and finite, while the set of extreme directions is empty. Any point $\bar{x}$ can be represented as a convex combination of extreme*

points, if and only if $\bar{x} \in X$, that is,

$$\begin{aligned} \bar{x} &= \sum_{j=1}^{l} \lambda_j \hat{x}_j \\ & \sum_{j=1}^{l} \lambda_j = 1 \\ & \lambda_j \geq 0 \quad j = 1, 2, \ldots, l \end{aligned}$$

In the general case, the representation theorem includes extreme directions, which is only necessary when we work with unbounded sets. A proof and more background theory on the theorem can be found in many textbooks on Linear Programming, including [BJS04]. The representation theorem can also be expressed on integer sets X, where the representation can be achieved using the relation between integer extreme points and integer extreme directions. For more information on Integer Linear Programming, Column Generation, Branch and Price and the Dantwzig-Wolfe Decomposition, we refer to [Van00]. This implies that the decomposition principles presented in the sequel also work for ILP's. For reasons of simplicity, we ignore the question whether we are dealing with a LP or ILP and focus on the procedure itself.

We will transform the *LRRP* using the representation theorem. First we choose the subsystems that we will represent using their extreme points. We select the subsystem $X = \{x_k | A_3 x_k \leq b_2\}$ and for each scenario $s \in S$ a subsystem $Y^s = \{y_j^s | A_4^s y_j^s \leq b_3^s\}$. We introduce u_k as a multiplier for the kth extreme points in X (so u_k will substitute λ_k in the representation theorem for set X), while v_j^s will be the multiplier for the jth extreme point in Y^s (so v_j^s will substitute λ_j in the representation theorem for set Y^s). Doing this, we get

$$\begin{array}{lll} \min & c^1 x + \sum_{s \in S} c_s^2 y^s & \\ \text{s.t.} & A_1^s x + A_2^s y^s \leq b_1^s & \forall s \in S \\ & A_3 x \leq b_2 & \\ & A_4^s y^s \leq b_3^s & \\ & \sum_k x_k u_k = x & \\ & \sum_k u_k = 1 & \\ & \sum_j y_j^s u_j^s = y^s & \forall s \in S \\ & \sum_j v_j^s = 1 & \forall s \in S \\ & u_k \geq 0 & \forall k \\ & v_j^s \geq 0 & \forall s \in S, \forall j \end{array}$$

Now let us apply some substitutions : since we have $\sum_k x_k u_k = x$ and $\sum_j y_j^s v_j^s = y^s$, we can remove the x and y^s vectors from the problem. We substitute all occurrences of x and y^s with our new representation. This yields the following:

$$\begin{array}{llll}
\min & c^1(\sum_k x_k u_k) + \sum_{s\in S} c_s^2(\sum_j y_j^s v_j^s) & & \\
\text{s.t.} & A_1^s(\sum_k x_k u_k) + A_2^s(\sum_j y_j^s v_j^s) & \leq b_1^s & \forall s \in S \\
& A_3(\sum_k x_k u_k) & \leq b_2 & \\
& A_4^s(\sum_j y_j^s v_j^s) & \leq b_3^s & \\
& \sum_k u_k & = 1 & \\
& \sum_j v_j^s & = 1 & \forall s \in S \\
& u_k & \geq 0 & \forall k \\
& v_j^s & \geq 0 & \forall s \in S, \forall j
\end{array}$$

From this from, we remove the $A_3(\sum_k x_k u_k) \leq b_2$ and $A_4^s(\sum_j y_j^s v_j^s) \leq b_3^s, \forall s \in S$ subsystems by decomposing them to sub-problems. We also use the distributive law from elementary algebra to tidy our objective and the recovery constraints. This way, we finally derive the Linear Separate Master Problem ($LSMP$)

$$\begin{array}{lllllll}
LSMP_{\min} & = & \min & \sum_k (c^1 x_k)u_k + \sum_{s\in S}\sum_j (c_s^2 y_j^s)v_j^s & & & \\
& & \text{s.t.} & \sum_k (A_1^s x_k)u_k + \sum_j (A_2^s y_j^s)v_j^s & \leq b_1^s & \forall s \in S & |S| \text{ dual vectors } \pi_s \\
& & & \sum_k u_k & = 1 & & \text{dual } \pi^X \\
& & & \sum_j v_j^s & = 1 & \forall s \in S & |S| \text{ duals } \pi_s^Y \\
& & & u_k \in \{0,1\} & & \forall k & \\
& & & v_j^s \in \{0,1\} & & \forall s \in S, \forall j &
\end{array}$$

Now that we have defined our master problem, we will also derive the necessary pricing problems. We will begin with the pricing problem for an initial solution. Now we will consider the system for set X we removed from our master problem, including its objective. This yields

$$\begin{array}{lll}
\min & \sum_k (c^1 x_k)u_k & \\
\text{s.t.} & A_3(\sum_k x_k u_k) & \leq b_2
\end{array}$$

Since we want to use column generation, we will adapt the objective of this system to take the duals of the master problem into account. In general, the reduced costs of a new column in the master problem are denoted by $c' - z'$, with c' the costs of the new

column and z' the product of the current dual vector of the master problem and the new column in the constraint matrix. In this case, a new column will have 1 in the row with the π^X dual and the extreme point itself for each of π_s dual vectors, so for a certain extreme point x_k, we have $z_k = \pi^X + \sum_{s\in S} \pi_s(A_1^s x_k)$ and since we enumerate over all possible x_k, we have $z' = \sum_k z_k u_k$. The resulting system that takes these reduced costs into account is

$$\begin{array}{lll} \min & \sum_k (c^1 x_k) u_k - \sum_k \sum_{s\in S} \pi_s (A_1^s x_k) u_k - \pi^X & \\ \text{s.t.} & A_3 (\sum_k x_k u_k) & \leq b_2 \end{array}$$

For practical purposes, we'd rather want a pricing problem that is concerned with the original decision variables for X. To achieve this, we use the representation theorem again. In the derivation of the master problem, we used the equation $\sum_k x_k u_k = x$ to substitute x for a representation in extreme points. This time, we will substitute the extreme point representation with the original decision variables. We also simplify the objective, which yields the final form of the Linear Separate Initial Pricing Problem $LSIPP$.

$$\begin{array}{rclll} LSIPP_{\min} & = & \min & (c^1 - \sum_{s\in S} \pi_s A_1^s) x - \pi^X & \\ & & \text{s.t.} & A_3 x & \leq b_2 \end{array}$$

This still leaves us with the Y^s systems for each scenario s. We will apply a similar procedure, starting with the basic system we have for each scenario $s \in S$.

$$\begin{array}{lll} \min & \sum_{s\in S} \sum_j (c_s^2 y_j^s) v_j^s & \\ \text{s.t.} & A_4^s (\sum_j y_j^s v_j^s) & \leq b_3^s \end{array}$$

Again, we must take the reduced costs into account if we want to use column generation. A recovery column for a scenario s will contain a 1 in the row that corresponds to the π_s^Y dual. In addition to that, the column will contain the extreme point in the rows that correspond to the π_s dual vector. Therefore, the reduces costs of a certain y_j^s vector are $c_s^2 y_j^s - \pi_s A_2^s y_j^s - \pi_s^Y$. We include this in our system to get

$$\begin{array}{lll} \min & \sum_j (c_s^2 y_j^s - \pi_s A_s^s) v_j^s & \\ \text{s.t.} & A_4^s (\sum_j y_j^s v_j^s) & \leq b_3^s \end{array}$$

Our last step is to apply the representation theorem again, this time using the $\sum_j y_j^s u_j^s = y^s$ equation. We rearrange the factors in a nice way and we get the Linear Separate Recovery Pricing Problem for scenario $s \in S$, $LSRPP^s$

$$
\begin{array}{rclll}
LSRPP^s_{\min} & = & \min & (c^2_s - \pi_s A^s_2) y^s - \pi^Y_s & \\
 & & \text{s.t.} & A^s_4 y^s & \leq b^s_3
\end{array}
$$

3.4.2. Combined Recovery Decomposition

Like the Separated Recovery Decomposition, the Combined Recovery Decomposition is derived from LRRP. Let us consider its form one more time.

$$
\begin{array}{rclllll}
LRRP_{\min} & = & \min & c^1 x + \sum_{s \in S} c^2_s y^s & & & \text{objective} \\
 & & \text{s.t.} & A^s_1 x + A^s_2 y^s & \leq b^s_1 & \forall s \in S & \text{recovery constraints} \\
 & & & A_3 x & \leq b_2 & & \text{initial variable constraints} \\
 & & & A^s_4 y^s & \leq b^s_3 & \forall s \in S & \text{scenario constraints}
\end{array}
$$

We need to rewrite this general form to be able to retrieve the Combined Recovery decomposition, by introducing some dummy variable vectors, $x'^s \forall s \in S$ and making sure they are always equal to variable vector x. This gives us the following form:

$$
\begin{array}{lllll}
\min & c^1 x + \sum_{s \in S} c^2_s y^s & & & \text{objective} \\
\text{s.t.} & x & = x'^s & \forall s \in S & \\
 & A^s_1 x + A^s_2 y^s & \leq b^s_1 & \forall s \in S & \text{recovery constraints} \\
 & A_3 x & \leq b_2 & & \text{initial variable constraints} \\
 & A^s_4 y^s & \leq b^s_3 & \forall s \in S & \text{scenario constraints}
\end{array}
$$

It is easy to see that this will not change the possible solutions or the objective of the problem. Now since we introduced variables that are equal to x, we can move some of the constraint matrices that work on x to our new variables. We will first apply this principle to the A^s_1 matrices, yielding the following form:

$$
\begin{array}{lllll}
\min & c^1 x + \sum_{s \in S} c^2_s y^s & & & \text{objective} \\
\text{s.t.} & x & = x'^s & \forall s \in S & \\
 & A^s_1 x'^s + A^s_2 y^s & \leq b^s_1 & \forall s \in S & \text{recovery constraints} \\
 & A_3 x & \leq b_2 & & \text{initial variable constraints} \\
 & A^s_4 y^s & \leq b^s_3 & \forall s \in S & \text{scenario constraints}
\end{array}
$$

This transformation is also acceptable, since the x vector will still be constrained by all A^s_1 matrices, due to the $x = x'^s$ constraints. Now we will apply the final transformation, where we move the A_3 matrix along the $x = x'^s$ constraints. This yields the final form, the Transformed linear recoverable robustness problem $TLRRP$:

$$
\begin{array}{llllll}
TLRRP_{\min} & = & \min & c^1x + \sum_{s\in S} c^2_s y^s & & & \text{objective} \\
 & & \text{s.t.} & x & = x'^s & \forall s \in S & \\
 & & & A^s_1 x'^s + A^s_2 y^s & \leq b^s_1 & \forall s \in S & \text{recovery constraints} \\
 & & & A_3 x'^s & \leq b_2 & \forall s \in S & \text{initial variable constraints} \\
 & & & A^s_4 y^s & \leq b^s_3 & \forall s \in S & \text{scenario constraints}
\end{array}
$$

Now let us consider the application of the representation theorem for bounded sets (Theorem 3.1) on the $TLRRP$. For each scenario $s \in S$, we will consider a subsystem $R_s = \{x'^s_p y^s_q | As_1 x'^s + A^s_2 y^s \leq b^s_1, A_3 x'^s \leq b_2, A^s_4 y^s \leq b^s_3\}$. We will use the representation theorem to represent a point $x'^s_p y^s_q$ using a multiplier w^s_{pq}. When we apply this to $TLRRP$, we get

$$
\begin{array}{llll}
\min & c^1x + \sum_{s\in S} c^2_s y^s & & \\
\text{s.t.} & x'^s & = x & \forall s \in S \\
 & A^s_1 x'^s + A^s_2 y^s & \leq b^s_1 & \forall s \in S \\
 & A_3 x'^s & \leq b_2 & \forall s \in S \\
 & A^s_4 y^s & \leq b^s_3 & \forall s \in S \\
 & \sum_{(p,q)} x'^s_p w^s_{pq} & = x & \forall s \in S \\
 & \sum_{(p,q)} y^s_q w^s_{pq} & = y^s & \forall s \in S \\
 & \sum_{(p,q)} w^s_{pq} & = 1 & \forall s \in S \\
 & w^s_{pq} & \geq 0 & \forall s \in S, \forall pq
\end{array}
$$

Now let us remove the x'^s and y^s vectors from the problem, using the equations $\sum_{(p,q)} x'^s_p w^s_{pq} = x$ and $\sum_{(p,q)} y^s_q w^s_{pq} = y^s$. We now get

$$
\begin{array}{llll}
\min & c^1x + \sum_{s\in S} c^2_s (\sum_{(p,q)} y^s_q w^s_{pq}) & & \\
\text{s.t.} & (\sum_{(p,q)} x'^s_p w^s_{pq}) & = x & \forall s \in S \\
 & A^s_1 (\sum_{(p,q)} x'^s_p w^s_{pq}) + A^s_2 (\sum_{(p,q)} y^s_q w^s_{pq}) & \leq b^s_1 & \forall s \in S \\
 & A_3 (\sum_{(p,q)} x'^s_p w^s_{pq}) & \leq b_2 & \forall s \in S \\
 & A^s_4 (\sum_{(p,q)} y^s_q w^s_{pq}) & \leq b^s_3 & \forall s \in S \\
 & \sum_{(p,q)} w^s_{pq} & = 1 & \forall s \in S \\
 & w^s_{pq} & \geq 0 & \forall s \in S, \forall pq
\end{array}
$$

For each scenario $s \in S$, we move the A_1^s, A_2^s, A_3 and $A3_4$ subsystems into a sub-problem. This leaves us with the Linear Combined Master Problem, $LCMP$.

$$\begin{array}{rcllll} LCMP_{\min} & = & \min & c^1 x + \sum_{s \in S} \sum_{(p,q)} (c_s^2 y_p^s) w_{pq}^s & & \\ & & \text{s.t.} & \sum_{(p,q)} w_{pq}^s = 1 & \forall s \in S & |S| \text{ duals } \pi^s \\ & & & \sum_{(p,q)} x_p w_{pq}^s = x & \forall s \in S & |S| \text{ dual vectors } \pi'^s \\ & & & w_{pq}^s \in \{0,1\} & \forall s \in S, & \\ & & & & \forall (p,q) & \end{array}$$

Now, we have a subsystem for each scenario that contains the initial and recovery systems.

$$\begin{array}{lll} \min & \sum_{(p,q)} (c_s^2 y_p^s) w_{pq}^s & \\ \text{s.t.} & A_1^s (\sum_{(p,q)} x_p'^s w_{pq}^s) + A_2^s (\sum_{(p,q)} y_q^s w_{pq}^s) & \leq b_1^s \\ & A_3 (\sum_{(p,q)} x_p'^s w_{pq}^s) & \leq b_2 \\ & A_4^s (\sum_{(p,q)} y_q^s w_{pq}^s) & \leq b_3^s \end{array}$$

Now let us consider the reduced costs. We will generate columns for each scenario $s \in S$, based on our pricing problem. Such a column will contain a 1 in the row of the corresponding π^s duals. The column will contain the $x_p'^s$ part of the extreme point. For a certain (p,q), we get a $z'_{pq} = \pi^s x_p^s + pi^s$. If we include the reduced costs in our system, we get

$$\begin{array}{lll} \min & \sum_{(p,q)} (c_s^2 y_p^s) w_{pq}^s - \sum_{(p,q)} pi^s x_p^s - \pi_s & \\ \text{s.t.} & A_1^s (\sum_{(p,q)} x_p'^s w_{pq}^s) + A_2^s (\sum_{(p,q)} y_q^s w_{pq}^s) & \leq b_1^s \\ & A_3 (\sum_{(p,q)} x_p'^s w_{pq}^s) & \leq b_2 \\ & A_4^s (\sum_{(p,q)} y_q^s w_{pq}^s) & \leq b_3^s \end{array}$$

Now we will transform this system into a pricing problem on the original decision variables, using the representation theorem again. We use $\sum_{(p,q)} x_p'^s w_{pq}^s = x$ and $\sum_{(p,q)} y_q^s w_{pq}^s = y^s$, to derive the Linear Combined Pricing Problem for scenario $s \in S$ ($LCPP^s$),

$$
\begin{array}{rclll}
LCPP^s_{\min} & = & \min & c^2_s y^s - \pi'^s x^s - \pi_s & \\
 & & \text{s.t.} & A^s_1 x'^s + A^s_2 y^s & \leq b^s_1 \\
 & & & A_3 x'^s & \leq b_2 \\
 & & & A^s_4 y^s & \leq b^s_3
\end{array}
$$

While this is a less straightforward application of the Dantzig-Wolfe decomposition, since the x vector remains in the master problem, the principle is exactly the same.

3.5. MiniMax Objective Functions

The normal objective functions are minimization and maximization. While these are sufficient to express expected costs objectives, they cannot express objective where we want to minimize our costs in the worst case scenario. For some problems it is easy to point out the worst case scenario (with the knapsack problem one can see that the worst case scenario is the scenario with the lowest value for the capacity constraint), but for some problems it would be nice to take such an objective in account.

There is a simple trick to express such an objective in a linear program by using a special variable and some additional constraint. In case of minimizing the costs of the worst case scenario, we create a variable $z_{\max}$, that will hold the highest costs of the moment. With the additional constraints we will move the values of the $c^2_s y^s$ costs into the variable. Since the initial costs are always the same, we will retain them in the objective. This way, our general form is adapted into the following:

$$
\begin{array}{rclll}
LRRP_{\substack{\min \\ \max}} & = & \min & c^1 x + z_{\max} & \\
 & & \text{s.t.} & z_{\max} & \geq c^2_s y^s \quad \forall s \in S \\
 & & & A^s_1 x + A^s_2 y^s & \leq b^s_1 \quad \forall s \in S \\
 & & & A_3 x & \leq b_2 \\
 & & & A^s_4 y^s & \leq b^s_3 \quad \forall s \in S
\end{array}
$$

Now the objective tells to give $z_{\max}$ the lowest possible value, while the constraints state that $z_{\max}$ should at least have the value of each separate $c^2_s y^s$. The result will be that $z_{\max}$ will have the value of the highest $c^2_s y^s$, which is exactly what we want.

In case of a maximization problem where we want to maximize our minimum profit, we can use the same trick by introducing a $z_{\min}$ variable. In this case the general form is altered in the following way:

$$
\begin{array}{rclll}
LRRP_{\substack{\max \\ \min}} & = & \max & c^1 x + z_{\min} & \\
 & & \text{s.t.} & z_{\min} & \leq c^2_s y^s \quad \forall s \in S \\
 & & & A^s_1 x + A^s_2 y^s & \leq b^s_1 \quad \forall s \in S \\
 & & & A_3 x & \leq b_2 \\
 & & & A^s_4 y^s & \leq b^s_3 \quad \forall s \in S
\end{array}
$$

Of course, changing the general form has a certain impact on both decompositions. In the case of the separate recovery decomposition, the value of the scenarios is both in the master problem and in the recovery recovery pricing problem. Moving $c^2_s y^s$ into

constraints will create additional duals to take into account during the pricing problem. In our original pricing problem, the value of $c_s^2 y^s$ became the costs of a new column of variable v_j^s. Let's call these costs c_s^j. When we alter the original master problem, we get the following:

$$
\begin{array}{llllll}
LSMP_{\substack{\min\\\max}} = & \min & \sum_k (c^1 x_k) u_k + z_{\max} & & & \\
& \text{s.t.} & \sum_j (c_s^2 y_j^s) v_j^s & \leq z_{\max} & \forall s \in S & |S| \text{ duals: } \pi_s^z \\
& & \sum_k (A_1^s x_k) u_k + \sum_j (A_2^s y_j^s) v_j^s & \leq b_1^s & \forall s \in S & |S| \text{ dual vectors } \pi_s \\
& & \sum_k u_k & = 1 & & \text{dual } \pi^X \\
& & \sum_j v_j^s & = 1 & \forall s \in S & |S| \text{ duals } \pi_s^Y \\
& & u_k \in \{0,1\} & & \forall k & \\
& & v_j^s \in \{0,1\} & & \forall s \in S, \forall j &
\end{array}
$$

Since the initial pricing problem $LSIPP$ does not contain the y variables or their prices, this problem remains unchanged (i.e. $LSIPP_{\substack{\min\\\max}} = LSIPP_{\min}$). Since the recovery pricing problems contain these values, we need to adapt them. This means that we need to multiply $c_s^2 y^s$ with the dual from the master problem. This gives us the following adaptation of the recovery pricing problems:

$$
\begin{array}{llll}
LSRPP^s_{\substack{\min\\\max}} = & \min & (-\pi_s^z c_s^2 - \pi_s A_s^2) y^s - \pi_s^Y & \\
& \text{s.t.} & A_4^s y^s & \leq b_3^s
\end{array}
$$

The nice thing about this modification is the fact that only the objective of the pricing problem changes. This implies that in all cases where we have an efficient algorithm for the pricing problem that has no assumptions on the costs of the pricing problem, we can keep using it when we reformulate our problem with a minimax objective.

The same property holds for the combined recovery decomposition. The process is more or less the same: we change the objective of the master problem into the following by adding a couple of constraints:

$$
\begin{array}{llllll}
LCMP_{\substack{\min\\\max}} = & \min & c^1 x + z_{\max} & & & \\
& \text{s.t.} & \sum_{(p,q)} (c_s^2 y_q^s) w_{pq}^s & \leq z_{\max} & \forall s \in S & |S| \text{ duals: } \pi_s^z \\
& & \sum_{(p,q)} w_{pq}^s & = 1 & \forall s \in S & |S| \text{ duals } \pi^s \\
& & \sum_{(p,q)} x_p w_{pq}^s - x & = 0 & \forall s \in S & |S| \text{ dual vectors } \pi'^s \\
& & w_{pq}^s & \in \{0,1\} & \forall s \in S, & \\
& & & & \forall (p,q) &
\end{array}
$$

The new duals π_s^z are added to the objective function of the pricing problems. Again, the constraints of the pricing problem remain unchanged.

$$\begin{array}{rcll} LCPP^s_{\substack{\min\\\max}} & = & \min & -\pi'_s x'^s - (\pi_s^z c_s^2) y^s - \pi_s \\ & & \text{s.t.} & A_1^s x'^s + A_2^s y^s \quad \leq b_1^s \\ & & & A_3 x'^s \quad \leq b_2 \\ & & & A_4^s y^s \quad \leq b_3^s \end{array}$$

3.6. Scenario Generation

When we consider minimax objectives, we can make the observation that in certain cases our objective value will be determined by a small number of scenarios. These are the scenarios that have the worst possible solution in the current situation. In certain cases it can be a good idea too start with only a restricted number of scenarios in the decomposition model, search for a solution and apply the recovery algorithm for the unconsidered scenarios on the initial solution found. If we get a scenario that yields a solution which is worse than the worst solution in our restricted model, we can add that scenario to the model, solve it with the solution we had as a starting point and repeat this process, until we cannot find a scenario that exceeds the worst solution so far.

Additionally, it is thinkable to use this approach in different situations. For example when only feasibility is our concern, we can use this approach to concentrate the process on scenarios that yield no feasible solution in the current situation, instead of all scenarios at once.

There is a clear relation between column generation and this approach: where column generation is an approach that tries to find a good solution by finding new variables that may improve the current solution, instead of considering all variables at once. Scenario generation tries to find a scenario that does not fit well to the current solution, instead of considering all scenarios at once.

4. Example Decompositions for Robustness Problems

4.1. A Classroom Problem Example

Now let us consider the classroom problem. We need to buy instant classroom-containers for a uncertain number of students. We may buy some classrooms initially for 3 units of money. We may take an option on classrooms for 2 units of money. If we know the exact number of students, we may buy classroom we took an option on for an additional 2 units of money. If we didn't take an option, we still may buy classroom for 10 units of money, after we know the exact number of students.

There is a probability of 0.6 that 110 students will enroll, a probability of 0.3 that 210 students will enroll and a probability of 0.1 that 300 students will enroll. Let us introduce a variable x_1 for the initial classrooms we buy and a variables x_2 for the number of options we take initially. We introduce variables y_1^s for the number of classrooms we buy using an option, while y_2^s is number of classrooms we buy in scenario $s \in S$. When we formulate this problem as an ILP, we get the following ILP:

$$
\begin{array}{llllllllll}
CP = & & & & & & & & & \\
\text{min} & 3x_1 & +2x_2 & +1.2y_1^1 & +6y_2^1 & +0.6y_1^2 & +3y_2^2 & +0.2y_1^3 & +1y_2^3 & \\
\text{s.t.} & 35x_1 & & +35y_1^1 & +35y_2^1 & & & & & \leq 110 \\
 & 35x_1 & & & & +35y_1^2 & +35y_1^2 & & & \leq 210 \\
 & 35x_1 & & & & & & +35y_1^3 & +35y_1^3 & \leq 300 \\
 & & x_2 & -y_1^1 & & & & & & \geq 0 \\
 & & x_2 & & & -y_1^2 & & & & \geq 0 \\
 & & x_2 & & & & & & -y_1^32 & \geq 0 \\
 & x_1, & x_2, & y_1^1, & y_2^1, & y_1^2, & y_2^2, & y_1^3, & y_2^3 & \in \{0 \ldots 10\}
\end{array}
$$

Since we have no real constraint on the initial solution, we will use the Combined Recovery Decomposition. First we derive the pricing problem, by filling in the $LCMP$ from Section 3.3. We introduce indices p, q on the possible combinations for the x and y^s variables. We introduce x_p^1 as the value of x_1 in solution p, while x_p^2 is the value of x_2 in solution p. This gives us the following Combined Master Problem:

$$
\begin{array}{llllll}
CP - LCMP = & \text{min} & 3x_1 + 2x_2 + \sum_{s \in S} \sum_{p,q} c_q^s w_{pq}^s & & & \\
 & & \sum_{p,q} w_{pq}^s & = 1 & \forall s \in S & \text{duals: } \pi^s \\
 & & \sum_{p,q} x_p^1 w_{pq}^s & = x_1 & \forall s \in S & \text{dual: } \pi_1 \\
 & & \sum_{p,q} x_p^2 w_{pq}^s & = x_2 & \forall s \in S & \text{dual: } \pi_2
\end{array}
$$

Since we have three scenarios, we have three Combined Pricing Problems associated with this problem. By filling in the $LCPP^s$ from Section 3.3 for each scenario $s \in S$, we get the following pricing problems:

$$
\begin{array}{rllllll}
CP-LCCP^1 = & \min & -\pi_1 x_1 & -\pi_2 x_2 & +1.2y_1^1 & +6y_2^1 & -\pi^1 \\
& \text{s.t.} & 35x_1 & & +35y_1^1 & +35y_2^1 & \leq 110 \\
& & & x_2 & -y_1^1 & & \geq 0 \\
& & x_1, & x_2, & y_1^1, & y_2^1 & \in \{0\dots 10\} \\
CP-LCCP^2 = & \min & -\pi_1 x_1 & -\pi_2 x_2 & +0.6y_1^2 & +3y_2^2 & -\pi^2 \\
& \text{s.t.} & 35x_1 & & +35y_1^2 & +35y_1^2 & \leq 210 \\
& & & x_2 & -y_1^2 & & \geq 0 \\
& & x_1, & x_2, & y_1^2, & y_2^2 & \in \{0\dots 10\} \\
CP-LCCP^3 = & \min & -\pi_1 x_1 & -\pi_2 x_2 & +0.2y_1^3 & +1y_2^3 & -\pi^3 \\
& \text{s.t.} & 35x_1 & & +35y_1^3 & +35y_1^3 & \leq 300 \\
& & & x_2 & -y_1^3 & & \geq 0 \\
& & x_1, & x_2, & y_1^3, & y_2^3 & \in \{0\dots 10\}
\end{array}
$$

This is just a simple example of the Combined Recovery Decomposition, on a rather easy problem. In the following sections, we will show the application of the framework to more well-known problems.

4.1.1. A Robust Knapsack Problem

Let us consider the classic Knapsack problem. We have some set of items, each with a unique index i from the set i and a given profit c_i and a given weight a_i. We also have a single capacity b. The problem states: what is the best possible subset of items with respect to their profits, that does not exceed capacity b with respect to their weights? It is often used as an example for an Integer Linear Program:

$$
\begin{array}{lll}
\max & \sum_{i \in I} c_i x_i & \\
\text{s.t.} & \sum_{i \in I} a_i x_i \quad \leq b & \\
& x_i \in \{0,1\} & \forall i \in I
\end{array}
$$

If we take a look at this problem, we can see that it satisfies the restrictable variable domains (RVD). If a variable is fixed to 0, we can remove that item from the problem and ignore it when we solve it. If we fix it to 1, we can reduce the capacity b by it's weight a_i, remove it from the problem and add its profit c_i to the total profit after we found the solution to the reduced problem Additionally, linear adjustable objectives (LAO), are also implied. If an item has a negative profit, we can remove it from the problem and ignore it. Otherwise, we just add or remove some constant from its c_i and solve the problem like normal.

Let us consider the example we mentioned earlier: the factory that has to make choices. In fact, this problem is a simple variant of the knapsack problem, where in the first place we have a certain high knapsack bound b^1 that we should fill. During recovery we will be presented with a lower knapsack bound b_s, depending on a scenario s. Let us assume that each scenario has a certain probability p_s and we want to maximize the expected

profit after recovery. We have a set of items I, where c_i is the profit of an item i and a_i is it's size. We have a vector x, that has a binary variable x_i for each item in I, telling if we should select it initially. We also have a vector y^s for each scenario s, that has a binary variable y_i^s for each item in I, telling we should keep item i from the initial solution in the recovery solution for scenario s. We get the following Linear Recoverable Knapsack Problem with Recovery by Removal ($LRKP - R$):

$$
\begin{array}{rcllll}
\text{LRKP-R} & = & \max & \sum_{i \in I} p_0 c_i x_i + \sum_{s \in S} \sum_{i \in I} p_s c_i y_i^s & & \\
& & \text{s.t.} & \sum_{i \in I} a_i x_i & \leq b & \\
& & & \sum_{i \in I} a_i y_i^s & \leq b_s & \forall s \in S \\
& & & x_i & \geq y_i^s & \forall i \in I, \forall s \in S \\
& & & x_i, y_i^s \in \{0,1\} & & \forall i \in I, \forall s \in S
\end{array}
$$

It is clear that this problem has linear recovery constraints. Additionally, the problem is almost always feasible: only if our capacity is negative, we cannot find a feasible solution, otherwise we can ignore all items and take an empty knapsack. This principle can be translated to a simple feasibility algorithm (FA). Also, we can express the recovery problem that may be solved by a recovery algorithm (RA) as follows: suppose we have some items selected initially and we want to find a recovery for a scenario s. Suppose the sum of the weights is initially larger than the capacity b_s. We can express this recovery problem as a single knapsack problem: select items from the items selected initially with maximum profit such that capacity b_s is not exceeded. If it is not larger, we can just take the initial solution as our recovery solution. We apply the separate recovery decomposition by defining χ_k as the kth initial solution added to our master problem and ϕ_j^s as the jth recovery solution for scenario s added to our master problem.

$$
\begin{array}{rclllll}
\text{LSMP-RKP-R} & = & \max & \sum_{k} c_k^X x'_k + \sum_{s \in S} \sum_{j} c^{Y_j^s} y'^s_j & & & \\
& & \text{s.t.} & \sum_{k} x'_k & = 1 & & \text{dual: } \pi \\
& & & \sum_{j} y'^s_j & = 1 & \forall s \in S & \text{duals: } \pi^s \\
& & & \sum_{k} \chi_{k,i} x'_k - \sum_{j} \phi_{j,i}^s y'^s_j & \geq 0 & \forall s \in S, \forall i \in I & \text{duals: } \pi_i^s \\
& & & x'_k, y'^s_j \in \{0,1\} & & \forall s \in S, \forall j, \forall k &
\end{array}
$$

$$\begin{array}{llll} \text{LSIPP-RKP-R} = & \max & \sum_{i\in I}(p_0 c_i - \sum_{s\in S}\pi_i^s)x_i - \pi & \\ & \text{s.t.} & \sum_{i\in I} a_i x_i & \leq b \\ & & x_i \in \{0,1\} & \forall i \in I \end{array}$$

$$\begin{array}{llll} \text{LSRPP-RKP-R}^s = & \max & \sum_{i\in I}(p_s c_i - \pi_i^s)y_i^s - \pi_s & \\ & \text{s.t} & \sum_{i\in I} a_i y_i^s & \leq b^s \\ & & x_i \in \{0,1\} & \forall i \in I \end{array}$$

Both pricing problems leave us with a single knapsack constraint. Both pricing problems are actually just knapsack problems with the same structure as the classic knapsack problem. Now let us apply the combined recovery decomposition. We introduce χ_j^s as the jth initial solution for scenario s added to the master problem. We get:

$$\begin{array}{llllll} \text{CMP-RKP-R} = & \max & \sum_{i\in I} p_0 c_i x_i + \sum_j c^{Y_j^s} y_j'^s & & & \\ & \text{s.t.} & \sum_j y_j'^s & = 1 & \forall s \in S & \text{duals: } \pi^s \\ & & \sum_j \chi_{j,i}^s y_j'^s & = x_i & \forall s \in S, \forall i \in I & \text{duals: } \pi_i^s \\ & & x_i \in \{0,1\} & & \forall i \in I & \\ & & y_j'^s \in \{0,1\} & & \forall s \in S, \forall i \in I & \end{array}$$

$$\begin{array}{lllll} \text{CPP-RKP-R}^s = & \max & \sum_{i\in I}(p_0 c_i - \pi_i^s)x_i + \sum_{i\in I} p_s c_i y_i^s - \pi^s & & \\ & \text{s.t.} & \sum_{i\in I} a_i x_i & \leq b & \\ & & \sum_{i\in I} a_i y_i^s & \leq b_s & \\ & & x_i & \geq y_i^s & \forall i \in I \\ & & y_i^s \in \{0,1\} & & \forall i \in I, \forall s \in S \end{array}$$

This is not a basic knapsack problem. We could apply the separate recovery decomposition to decompose this pricing problem into two separate problems with the separate recovery decomposition, but a nicer approach is to use a specific algorithm: a Dynamic Programming approach is presented in Section 5.3.5.

4.2. A Robust Weighted Independent Set Problem

Now let us consider the basic weighted Independent Set problem. Given a graph G with node set V and an edge set E, i.e. $G = (V, E)$ and a weight function w that gives the

weight of a certain node $v \in V$, we are asked to find a subset of nodes with a maximum sum of the weights of the nodes and the constraint that if there is an edge $(u, v) \in E$, we may not have both node u and node v in the set. If we express this problem as an Integer Linear Program, we get:

$$\begin{array}{rclll} \text{LWIS} & = & \max & \sum_{v \in V} w(v)x_v & & \\ & & \text{s.t.} & x_u + x_v & \leq 1 & \forall (u,v) \in E \\ & & & x_v \in \{0,1\} & & \forall v \in V \end{array}$$

Let us consider restrictable variable domains (RVD). When we fix the variable x_v of a node v to 1, we cannot select any neighbors, which implies that we can remove node v and all its neighbors from the problem. If we fix the variable x_v to 0, we can just remove that node from the problem. Again, the linear adjustable objectives (LAO) are rather trivial. If a certain node v has a negative profit $w(v)$, we can just ignore it, or remove it from the problem. Otherwise, we can easily change or add a factor to a certain $w(v)$ to derive a new weighted independent set problem.

Now let us extend this basic problem to a problem with uncertainty. We will consider a graph G with a fixed node set V and weights $w(v) \in \mathbb{Z}$. We will use the conference organization example from the introduction in Section 1.1. We define a initial edge set E^1 and realized edge sets E^2_s, such that E^1 is a subset of each realized edge set E^2_s. So each edge set E^2_s introduces new edges with respect to E^1. Now let us find a maximum weighted independent set as an initial solution, such that when we get a realized edge set we may only remove nodes from the weighted independent set to become feasible. In a sense looks a bit like the structure of the knapsack problem. We have an initial binary variable vector x that tells us for each $v \in V$ if that node is selected initially. We also have a binary variable vector y^s for each scenario s that tells us for each node if it remains selected in the recovery solution for scenario s. We have a probability p_s for each scenario s. We get the following program:

$$\begin{array}{rclll} \text{LRWIS-R} & = & \max & \sum_{v \in V} p_s w(v) y^s_v & & \\ & & \text{s.t.} & x_u + x_w & \leq 1 & \forall (u,w) \in E^1 \\ & & & y^s_u + y^s_w & \leq 1 & \forall (u,w) \in E^2_s, \forall s \in S \\ & & & x_v & \geq y^s_v & \forall s \in S, \forall v \in V \\ & & & x_v, y^s_v \in \{0,1\} & & \forall v \in V, \forall s \in S \end{array}$$

Again, the linear recovery constraints (LRC) are clearly implied by the formulation. Additionally, feasibility is not a very large issue: we only have infeasibility in a case where two nodes that are neighbors are fixed to 1 in the initial or recovery variable vectors or a case where a node is fixed to 1 in the recovery vector and to 0 in the initial vector. In other words: if we put all unrestricted variables to 0 and our solution remains infeasible, the current problem is infeasible. Now let us apply the separate recovery decomposition. The following integer linear programs are derived:

$$
\begin{array}{llllll}
\text{LSMP-WIS-R} & = & \max & {}_{s\in S}\sum_j (c_j^{Y^s} y_j'^s & & \\
& & \text{s.t.} & \sum_k x_k' & = 1 & \text{dual: } \pi^X \\
& & & \sum_j y_j'^s & = 1 & \forall s \in S \quad \text{duals: } \pi_s^Y \\
& & & \sum_k \chi_(k,v) x_k' & \geq \sum_j (\phi_{j,v}^s y_j'^s) & \forall v \in V, \forall s \in S \quad \text{duals: } \pi_s^v
\end{array}
$$

$$
\begin{array}{llll}
\text{LSIPP-WIS-R} & = & \max & -\sum_{v\in V}\sum_{s\in S} \pi_s^v x_v + \pi^X \\
& & \text{s.t.} & x_u + x_v \leq 1 \quad \forall (u,v) \in E^1 \\
& & & x_v \in \{0,1\} \quad \forall v \in V
\end{array}
$$

$$
\begin{array}{llll}
\text{LSRPP-WIS-R}^s & = & \max & \sum_{v\in V} (p_s w(v) - \pi_s^v) y_v^s - \pi_s^Y \\
& & \text{s.t.} & y_u^s + y_v^s \leq 1 \quad \forall (u,v) \in E_s^2 \\
& & & y_v^s \in \{0,1\} \quad \forall v \in V
\end{array}
$$

Again, these pricing problems follow the structure of the weighted independent set problem. This result is similar to the result of the decomposition of the knapsack extension. This is not really surprising, since both problems deal with subset selection and both problems have the same recovery constraints. This implies that the interaction between the master and pricing problems is very similar. In case of the combined recovery decomposition, we will also see something that is similar to the combined recovery decomposition of the knapsack problem. We use the same conventions we used with the separate recovery decomposition.

$$
\begin{array}{llllll}
\text{LCMP-WIS-R} & = & \max & \sum_{s\in S}\sum_j c_j^{Y^s} y_j'^s & & \\
& & \text{s.t.} & \sum_j y_j'^s & = 1 & \forall s \in S \quad \text{duals: } \pi_s \\
& & & \sum_j \phi_{j,v}^s y_j'^s & = x_v & \forall s \in S, \forall v \in V \quad \text{duals: } \pi_v^s
\end{array}
$$

$$
\begin{array}{lllll}
\text{LCPP-WIS-R}^s & = & \max & \sum_{v\in V} p_s w(v) y_v^s - \sum_{v\in V} \pi_v^s x_v - \pi_s & \\
& & \text{s.t.} & x_u + x_v \leq 1 & \forall (u,v) \in E^1 \\
& & & y_u^s + y_v^s \leq 1 & \forall (u,v) \in E_s^2 \\
& & & x_v \geq y_v^s & \forall v \in V \\
& & & x_v \in \{0,1\} & \forall v \in V \\
& & & y_v^s \in \{0,1\} & \forall v \in V
\end{array}
$$

The resulting pricing problem asks for a initial and a recovery solution for a single scenario, which is not a classic independent set problem. However, there is a way to reduce this problem to am independent set problem. Let us build a new graph. For each node v in the original graph, we will add a node v' and a node v'' to the new graph, where a selection of v' in the independent set implies that we select v only initially, while selection of v'' implies that we select v in both initially and after recovery. Clearly, we may not select v' and v'' at the same time, so we add an edge (v', v'') to our new graph. The weight of v' in our independent set problem will be $-\pi_v^s$ and the weight of v'' will be $-w(v) - \pi_v^s$.

Now we will consider all edges (u, v) in E^1. Since these edges will also be in E_s^2, we know that we may never select any combination of v', v'', u' and u''. So we add the edges (v', u'), (v', u''), (v'', u') and (v'', u'') to the new graph, completing the clique between the nodes. For each edge (u, v) that is in E^1 but not in E_s^2, we can select u' and v' together, but not u' and v'', v' and u'' or v'' and u''. We add the edges (u', v''), (v', u'') and (v'', u'') to the graph, still allowing both u' and v' in the independent set.

4.3. A Minimax Example: Demand Robust Shortest Path

Let us consider the shortest path problem: given a graph $G = (V, E)$ and weights $w_{(u,v)}$ on the edges $(u, v) \in E$, given a source node $a \in V$ and a sink node $b \in V$, what is the shortest path from a to b. We will extend this problem to the demand robust shortest path problem, after we introduce a predicate $\mathit{pathset}(a, b, X)$.

Definition The predicate $\mathit{pathset}(a, b, X)$ with $a \in V, b \in V$ and $X \subseteq E$ is true if and only if the set X contains a path from a to b.

Now let us introduce binary variables $x_{(u,v)}$ for each edge $(u, v) \in E$. By using set builder notation, we can easily describe a relation between these variables and a set of edges: the expression $\{(u, v) \in E : x_{(u,v)} = 1\}$ contains all edges for which the corresponding binary variable is set to 1. Now let us introduce a more formal definition of the shortest path problem.

$$\begin{aligned} \text{LSPP} \quad = \quad \min \quad & \sum_{(u,v)\in E} w_{(u,v)} x_{(u,v)} \\ \text{s.t.} \quad & \mathit{pathset}(a, b, \{(u, v) \in E | x_{(u,v)} = 1\}) \end{aligned}$$

Now let us extend this to the demand robust shortest path problem, as described by [DGRS05]. In this extension of the shortest path problem, there are multiple scenarios $s \in S$ that each define a separate sink b_s and a factor f_s. Now we can buy each edge (u, v) initially for $w_{(u,v)}$ or when we recover for a scenario s for $f_s w_{(u,v)}$. Now we want to minimize the maximum total costs Let us introduce binary variables $y_{(u,v)}^s$ for each scenario $s \in S$ and each edge $(u, v) \in E$. Let us consider the formal description:

$$
\begin{aligned}
\text{DRSPP} \;=\; \min \quad & \sum_{(u,v)\in E} w_{(u,v)} x_{(u,v)} + z_{\max} \\
\text{s.t.} \quad & \sum_{(u,v)\in E} f_s w_{(u,v)} y^s_{(u,v)} \leq z_{\max} \quad && \forall s \in S \\
& \mathit{pathset}(a, b_s, \{(u,v) \in E | x_{(u,v)} = 1 \vee y^s_{(u,v)} = 1\}) && \forall s \in S
\end{aligned}
$$

In this case, the RVDs are quite trivial: fixing a variable to 1 can yield two results: if it is not in the optimal solution, the edges that are will be selected in addition to the variable. If it is in the optimal solution, nothing changes. Fixing a variable to 0 yields the same result as removing it from the graph. Feasibility can be checking by looking at the connectivity of the graph between a and each b_s - if a and a single b_s are unconnected, the problem has become infeasible. The linear adjustable objectives are also rather trivial, since we can simply adjust the weights of each variable.

However, linear recovery constraints are a problem in this case. While it may be possible to express the *pathset* predicate using linear constraints, it is not a very elegant way to look at the problem. This implies we will use the combined recovery decomposition, since it does not require the linear recovery constraints. When we apply this decomposition, we will consider χ^s_k as a solution vector to a binary variable y'^s_k as a set of edges, for the sake of simplicity.

$$
\begin{aligned}
\text{LCMP-DRSPP} \;=\; \min \quad & \sum_{(u,v)\in E} w_{(u,v)} x_{(u,v)} \quad + z_{\max} \\
\text{s.t.} \quad & \sum_k y'^s_k c^{Y^s}_k \leq z_{\max} && \forall s \in S && \text{duals: } \pi^z_s \\
& \sum_k y'^s_k = 1 && \forall s \in S && \text{duals: } \pi_s \\
& \sum_k [(u,v) \in \chi^s_k] y'^s_k = x_{(u,v)} && \forall (u,v) \in E, \forall s \in S && \text{duals: } \pi^s_{(u,v)}
\end{aligned}
$$

Now, the corresponding pricing problem for a given scenario $s \in S$ has the following form.

$$
\begin{aligned}
\text{CPP-DRSPP}^s \;=\; \min \quad & -\sum_{(u,v)\in E} \pi^s_{(u,v)} x_{(u,v)} - \pi^z_s \sum_{(u,v)\in E} w_{(u,v)} f_s y^s_{(u,v)} \\
\text{s.t.} \quad & \mathit{pathset}(a, b_s, \{(u,v) \in E | x_{(u,v)} = 1 \vee y_{(u,v)} = 1\})
\end{aligned}
$$

While the pricing problem is close to the shortest path problem we presented originally, we have two binary decisions variables for each edge $(u,v) \in E$. However, we can fix this using some preprocessing and certain observations about the pricing problem:

- We will always select a variable that has negative total cost. We can consider the costs of these variables as 0 while solving the shortest path problem, as long as we fix them to 1 and take their costs into consideration after we found a path.

- Now all variables have non-negative costs. Since selecting the $x_{(u,v)}$ or $y^s_{(u,v)}$ variable of a certain edge $(u,v) \in E$ will only give us an advantage if they actually are on the path we are looking for and since we only need one of the two, we will always take the cheapest. The other variable can be removed from the problem, because it will never be selected.

After we apply these observations during a preprocessing phase, we have a single variable for each edge and all these variables have non-negative costs. The resulting problem is a single shortest path problem and since we have no negative cost edges, they can be solved by using Dijkstra's Shortest Path algorithm [Dij59].

4.4. Recoverable Robust Network Flow

The work in this section is the result of joint work with Thomas van Dijk, with the idea to apply this approach to the Integer Maximum Flow in Wireless Sensor Networks with Energy Constraint [BTvDvL08].

The Maximum Flow Problem is a classic problem in Combinatorial Optimization and Operations Research, of which the history is discussed in [Sch02]. Methods to solve Maximum Flows problems are discussed in many textbooks on algorithms, including [CSRL01]. The Maximum Flow Problem is concerned with finding a maximum flow in a graph $G = (V, E)$. Give a source $i \in V$, a sink $j \in V$ and a capacity for each edge $c_{uv} \forall (u,v) \in E$. If we introduce a variable f_{uv} for each edge $(u,v) \in E$, we can write this problem as the following linear program:

$$
\begin{aligned}
\text{MFP} \quad = \quad \max \quad & \sum_{(i,v)\in E_i} f_{iv} & & \\
\text{s.t.} \quad & f_{uv} & \leq c_{uv} \quad & \forall (u,v) \in E \\
& \sum_{(u,v)\in E_v} f_{uv} - \sum_{(v,w)\in E_v} f_{vw} & = 0 \quad & \forall v \in V, v \neq i, v \neq j
\end{aligned}
$$

We may note that this linear program has a totally unimodular constraint matrix, which implies that the optimal solution to this linear program will be integer if the capacities of the edges are integer. For more information on linear programming and totally unimodular matrices we refer to [BJS04]. Now we extend this MFP to a Recoverable Robust Maximum Flow Problem with Recovery by Removal. We assume that the capacities of the edges may become lower in each scenario $s \in S$ and we may only remove flow from the network to adhere to the new capacities. If we formulate this problem using linear programming, we get

$$
\begin{array}{rclll}
\text{RRMFP-R} & = & \max & p_0 \sum_{(i,v)\in E_i} f_{iv} + \sum_{s\in S} p_s \sum_{(i,v)\in E_i} f_{iv}^s & \\
& & \text{s.t.} & f_{uv} & \leq c_{uv} \quad \forall (u,v) \in E \\
& & & \sum_{(u,v)\in E} f_{uv} - \sum_{(v,w)\in E} f_{vw} & = 0 \quad \forall v \in V, v \neq i, v \neq j \\
& & & f_{uv}^s & \leq c_{uv}^s \quad \forall s \in S, \forall (u,v) \in E \\
& & & \sum_{(u,v)\in E} f_{uv}^s - \sum_{(v,w)\in E} f_{vw}^s & = 0 \quad \forall s \in S, \forall v \in V, v \neq i, v \neq j \\
& & & f_{uv} - f_{uv}^s & \geq 0 \quad \forall s \in S, \forall (u,v) \in E
\end{array}
$$

Let us apply the Separate Recovery Decomposition to this problem. We introduce $\mathcal{F}$ as the set of possible flows, with k an index on these flows. We also introduce an indicator g_k^{uv} that gives the flow over edge (u,v) in the flow with index k from the set $\mathcal{F}$. Using the scheme of the $LSMP$ from Section 3.4.1, we get the following master problem:

$$
\begin{array}{rcllll}
{}^{RRMFP-R}_{LSMP} & = & \max & \sum_{k\in\mathcal{F}} p_0 c_k x_k + \sum_{s\in S} p_s \sum_{k\in\mathcal{F}} c_k y_k^s & & \\
& & \text{s.t} & \sum_{k\in\mathcal{F}} x_k & = 1 & \text{dual: } \pi_0 \\
& & & \sum_{k\in\mathcal{F}} y_k^s & = 1 \quad \forall s \in S & \text{duals: } \pi_s \\
& & & \sum_{k\in\mathcal{F}} g_k^{uv} x_k - \sum_{k\in\mathcal{F}} g_k^{uv} y_k^s & \geq 0 \quad \forall s \in S, \forall (u,v) \in E & \text{duals: } \pi_s^{uv}
\end{array}
$$

This master problem leaves us with the basic scenario duals and a dual value for each edge for each scenario. When we derive the pricing problems in the same way as in Section 3.4.1, we get the following Separate Initial and Separate Recovery Pricing Problems.

$$
\begin{array}{rclll}
{}^{RRMFP-R}_{SIPP} & = & \max & p_0 \sum_{(i,v)\in E_i} f_{iv} - \sum_{(u,v)\in E} \sum_{s\in S} \pi_s^{uv} f_{uv} - \pi_0 & \\
& & \text{s.t.} & f_{uv} & \leq c_{uv} \quad \forall (u,v) \in E \\
& & & \sum_{(u,v)\in E_v} f_{uv} - \sum_{(v,w)\in E_v} f_{vw} & = 0 \quad \forall v \in V, v \neq i, v \neq j
\end{array}
$$

$$
\begin{array}{rclll}
{}^{RRMFP-R}_{SRPP^s} & = & \max & p_s \sum_{(i,v)\in E_i} f_{iv}^s + \sum_{(u,v)\in E} \pi_s^{uv} f_{uv}^s - \pi_s & \\
& & \text{s.t} & f_{uv}^s & \leq c_{uv}^s \quad \forall (u,v) \in E \\
& & & \sum_{(u,v)\in E_v} f_{uv}^s - \sum_{(v,w)\in E_v} f_{vw}^s & = 0 \quad \forall v \in V, v \neq i, v \neq j
\end{array}
$$

We must observe that the objective of these pricing problems has changed with respect to the MFP. While the MFP is concerned with finding a maximum flow, this objective becomes distorted in the pricing problem. However, we may observe that the constraints of these pricing problems follow the exact same structure as the MFP. Therefore, these pricing problems have a totally unimodular constraint matrix as well and will yield integer solution when we use a linear programming solver. In addition to that, these pricing problems may be solved with algorithms for the Min-Cost Flow problem, such as presented by [Coo98] or [AMO93].

5. Recoverable Robustness and the Knapsack Problem

5.1. Introduction to Knapsack Problems

In the introduction to robustness in Section 1.1 we discussed the repair factory example and showed that it could be modeled as a knapsack problem in Section 4.1.1. We will now take a look at the Knapsack problem and its variants extended with robustness. The basic knapsack Problem (KP), sometimes called the 0-1 Knapsack Problem, is concerned with a set of items I with weights a_i and profits c_i and a single capacity b. The problem states that we should find a subset of I in such a fashion that we maximize our profit (the sum of the c_i's in the subset), while the weights don't exceed the capacity. The problem is often used as a simple example of an Integer Linear Program.

$$\begin{aligned} \max & \sum_{i \in I} c_i x_i & \\ \text{s.t.} & \sum_{i \in I} a_i x_i \leq b & \\ & x_i \in \{0, 1\} & \forall i \in I \end{aligned}$$

The binary variables x_i represent the choice to take an item in the subset or not. Many variants of the knapsack problem exist. An example is the bounded knapsack problem, that puts a certain limit on the amount of times an item can be taken (in an integer linear programming perspective this would change the variable constraint to $x_i \in \{0 \ldots n_i\}$ with n_i the number of copies for item i). Another example is the unbounded knapsack problem, where we may take an unlimited amount of each item (again, from an integer linear programming perspective this would change the variable constraint to $x_i \in \mathbb{N}$). We can examine the regular Knapsack Problem without loss of generality, since we can express these variants by adding each item multiple times to that problem.

Of course, there are many cases where the problem has a trivial solution. If we have a problem instance where we can choose all available items without exceeding the capacity, it is clear this solution is optimal. Any item with negative profit can be ignored, since it will only make a solution worse and we have no obligation to choose it. If we have an item that has a weight a_i that is larger than the maximum capacity b, we will never be able to choose it, so it can be ignored.

A special case of the Knapsack Problem is the variant where the profit of an item is equal to its weight, i.e. $\forall i \in I : a_i = c_i$. This variant is called the Subset Sum problem (or SSP). This variant has historic significance since it was one of Karp's 21 NP-complete problems as presented in [Kar72] and thus one of the first problems shown to be NP-complete. While Karp called it the Knapsack Problem, it is nowadays more common to refer to it as the Subset Sum problem, since his formulation was "INPUT: $(a_1, a_2, \ldots, a_r, b) \in \mathbb{Z}^{n+1}$, PROPERTY: $\sum a_j x_j = b$ has a $0 - 1$ solution". It is also relevant in practice where certain partitioning problems arise, for example when we want to schedule jobs over two machines in such a fashion that the time difference between the start and finish of the execution is minimized.

Another variant of the Knapsack Problem is the Cardinality Constrained Knapsack Problem (or kKP), which is discussed in [CKPP98]. The kKP has an additional constraint, stating that the maximum number of items in the solution may not exceed a certain value k (which corresponds to adding $\sum_{i \in I} x_i \leq k$ as a constraint to the ILP-formulation). Small variations of this constraint also exist, like the *exact k-item knapsack problem* (or E-kKP), which states that the solution should have exactly k items (or $\sum_{i \in I} x_i = k$).

Another variant is the Precedence Constrained Knapsack Problem (or PCKP), which is discussed by [JN83]. It allows us to define precedence constraints on the items. A precedence constraint states that we may only choose a certain item, if its predecessors are also chosen. The graph that expresses these constraints is called the precedence graph. If we have an arc (i, j) in the arc set E of the precedence graph, we must choose i before choosing j (this corresponds to having an additional constraint $x_i \geq x_j \forall (i, j) \in E$ in the linear programming formulation). Other variants are discussed in the textbook about Knapsack Problems [KPP04].

While, the exact Subset Sum problem is NP-complete (and the Knapsack problem is NP-hard), it is perhaps the easiest NP-complete problem and there are algorithms that can solve many practical cases quite efficiently. There is a straightforward application of Dynamic Programming for the SSP which can solve a problem in $O(nb)$ time (Section 5.3.1), where n is the amount of items and b the capacity. Since b is a single parameter in the input of the problem, we may only consider a $O(nb)$ time algorithm to be polynomial if we consider the input to be in unary encoding. If we consider the input in binary encoding, the algorithm doesn't run in polynomial time with respect to the input size. Something similar holds for the cardinality constrained knapsack problem, which has a relatively efficient algorithm from [CKPP98] which takes $O(nkb)$ time if the cardinality of the knapsack is constrained by k. An algorithm that runs in polynomial time in case of unary encoding, but in exponential time in case of a binary encoding, is called a pseudo-polynomial time algorithm.

However, not all NP-complete problems have pseudo-polynomial algorithms. The problems that do are called *weakly* NP-hard, while the problems that don't are called *strongly* NP-hard. The notion of pseudo polynomial time algorithms and NP-hardness in the weak or strong sense was introduced by [JN83]. An example of a *strongly* NP-hard problem is the Knapsack Problem with Precedence Constraint on arbitrary graphs. However, if we know the structure of the graph to be a tree, the problem becomes *weakly* NP-hard, since there is known to be an algorithm that solves this problem in $O(nU)$ time, where U is an upper-bound on the value of the solution (and a very poor upper-bound can be calculated by taking $\sum_{i \in I} c_i$, which yields a pseudo-polynomial time algorithm).

5.2. Dynamic Programming for KP and related problems

5.2.1. Dynamic Programming for KP

The easiest Dynamic Program to solve the basic Knapsack Problem is the classic Bellman Recurrence [Bel57]. Our state variable $A(i, w)$ reflects this recurrence. The value of $A(i, w)$ is the best possible profit for a knapsack with weight exactly w containing a

subset of items with indices j such that $j \leq i$.

$$\begin{array}{lcll} A(i,0) & = & 0 & \\ A(0,w) & = & -\infty & \text{for } w \neq 0 \\ A(i,w) & = & \max \begin{cases} A(i-1,w) \\ A(i-1,w-a_i)+c_i \end{cases} & \end{array}$$

The first line of the recurrence states that the value of a knapsack with size 0 has value 0 - this is the empty knapsack. We assume the indices on the items start at 1, so if $i = 0$, we don't consider any items. Now if we don't consider any items, the only possible knapsack is the empty knapsack, which has size 0. It is impossible to create any other knapsack, so we give such a knapsack the value $-\infty$. Now if we may consider a new item i, we have two options for each fixed size w: we can either take the best solution so far for the size w (ignoring the new item) or we can add it to the best possible knapsack of size $w - a_i$.

Since the optimality of $A(i, w)$ only depends on $A(i-1, w)$ and $A(i-1, w-a_i)$, the problem of finding the optimal $A(i, w)$ is divided into two smaller sub-problems. This implies the correctness of the recurrence, but a more detailed proof can be found in Appendix C.1.

If we use memorization we can create a table for all entries of $A(i, w)$, which has $|I|b$ entries. Since each entry can be calculated in constant time if its two dependencies have been calculated, the algorithm runs in $O(|I|b)$ time. We can use the principle that we only need the column $A(i-1, w)$ to calculate the column $A(i, w)$; which implies that we only need to store the previous and current column, reducing our memory costs to $O(b)$. Clearly, this approach only gives the value of the solution, but not the solution itself. If we keep all values of $A(i, w)$ in memory, we can backtrack trough the table by determining whether $A(i, w) = A(i-1, w)$ or $A(i, w) = A(i-1, w-a_i) - c_i$. In the first case, we ignore item i, in the second case we know we must add item $i-1$ to our solution. We can then backtrack further from either $A(i-1, w)$ or $A(i-1, w-a_i)$, depending on whether i was ignored or added.

Another approach is to keep track of the items added during the construction of each column: we store the current item-set for each cell in the current column in memory. Of course, this approach uses $O(|I|b)$ memory, just like keeping the entire table in memory. However, when we work with linked lists, it is very probable that we don't use all memory, since not all values for $A(i, w)$ will use all items.

5.2.2. Balanced Dynamic Programming for SSP

As introduced in [Pis99], we can make the observation that in case of the SSP (and also in case of the KP), if the sum of the sizes of all items is greater than the capacity b, the size of the feasible solution with total size closest to b must lie somewhere between $b - a_{\max}$ and b, with a_{max} the greatest size of a single item. Clearly, if the solution would have a greater size, it is infeasible. On the other hand, if it is smaller, we can add an item without becoming infeasible, thus improving the value of the solution.

There are advanced techniques for the SSP and KP that use this principle. Let us consider the split solution to a KP or SSP.

Split Solution for KP Let us order the items in our item-set I by descending ratios, such that $\frac{c_i}{a_i} \geq \frac{c_{i+1}}{a_{i+1}}$. Let us define the split item as the item with index $\hat{j}$, such that

$$\sum_{i=1}^{\hat{j}-1} a_i \leq b \text{ and } \sum_{i=1}^{\hat{j}} a_i > b$$

In other words, when we add items with increasing indices to our knapsack, item $\hat{j}$ is the first item that causes an overflow in the capacity b. Now we define the split solution $\hat{x}$ as $\hat{x} = \{1, 2, \ldots, \hat{j} - 1\}$.

Let us start with the split solution $\hat{x}$ with size $\hat{w}$ and $\hat{j}$ the index of the split item. The split solution contains all items with indices $i < \hat{j}$. We will discuss a technique that derives infeasible and feasible states from the $\hat{x}$ state, such that these states have total size in the range $b - a_{\max}, \ldots, b + a_{\max}$. If such a state has a size in the range $\{b - a_{\max}, \ldots, b\}$, we may add an item to it. If it has a size in the range $\{b + 1, \ldots, b + a_{\max}\}$, we must remove an item from it.

Now let us define a state variable (k, i, w) with k an index in the range $\hat{j}, \ldots, |I|$ which represents the last item considered for addition, w a size in the range $b - a_{\max}, \ldots, b + a_{\max}$ which represents the size of the current solution and i an index in the range $1, \ldots, \hat{j} - 1$ which represents the highest index that has not been considered for removal from the current solution. We require that only items that were originally in $\hat{x}$ can be removed, by stating that i should be in the range $1, \ldots, \hat{j} - 1$. Additionally, we will require that after we remove an item i with from a certain state, no items with an index i' such that $i' > i$ may be removed from that state for a larger k. So, for example, if $\hat{j} = 6$ and we need to remove items 2 and 4 from our split solution, we must first remove item 4 from the corresponding state before removing item 2, since our requirement states that we may not remove item 4 after we have removed item 2.

Not let us consider two states (k_1, i_1, w_1) and (k_2, i_2, w_2) such that $w_1 = w_2$, $k_1 = k_2$ and $i_1 > i_2$. Since we consider the SSP and both solutions have the same size, we are only concerned with the states that can be derived from them. Since $i_1 > i_2$, we have considered less items for removal in state 1, which implies that we have more freedom to remove items from state 1 than from state 2. Clearly, a state where we remove item i_1 and get a new state with total size $w_1 - a_{i_1}$ cannot be derived from state 2, since item i_1 has either been removed from that state already, or an item with a lower index has been removed which implies that we may not remove item i_1 any more. This means that state 1 dominates state 2 and we only have to memorize state 1 in this case.

Now for each possible index k, we will create a table that holds a state for each value of w such that $b - a_{\max} \leq w \leq b + a_{\max}$. From such a table for index k, we will construct the table for index $k + 1$. First, we copy all states in the table for index k to the table for $k + 1$, modifying index k in each state to $k + 1$. This is acceptable, since this copying operation represents the situation where we don't use item $k + 1$ in our solution. After

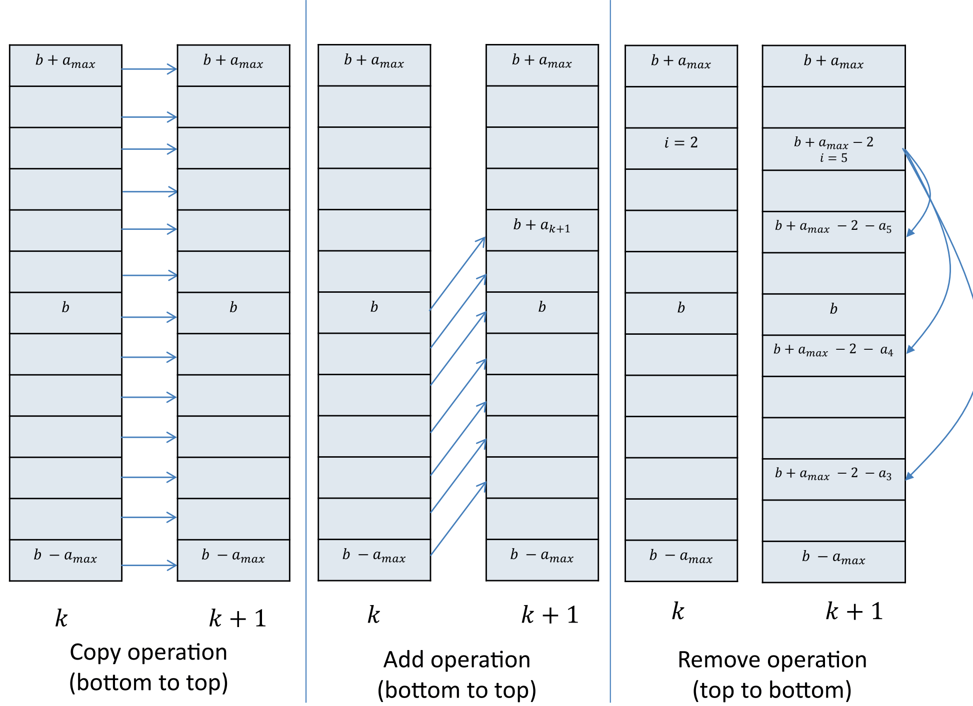


Figure 4: Example of constructing a table $k + 1$ from table k

the copying operation, we will perform two scanning phases through the table: one phase where we will add item $k + 1$ to states that have $w \leq b$ and a second phase where we remove items from states that have $w > b$. If we have a state with $w = b$, we are done (since no SSP solution will be better than an item-set with total weight b).

The copy operation presented above is very straightforward. After we have copied all old states, we will consider the states (k, i, w) with $w = b - a_{\max} + 1, \ldots, b$ and add a state $(k + 1, i, w + a_{k+1})$ for each w to the table for $k + 1$, if the generated states are not dominated by the current state in the table for $k + 1$. If the new state is dominated, we just retain the old state in the table. Both the copy and addition operations are illustrated in Figure 4.

After we have added our items, we will consider the infeasible states. We will now consider the states $(k + 1, i, w)$ with w from $b + a_{\max}$ down to $b + 1$. If the table for k contains a state (k, i', w), we will consider each index j such that $i' < j \leq i$ as a potential item for removal and generate states $(k + 1, j, w - a_j)$, which implies that we try to remove each item that hasn't been removed from a state with size w before. Now if there is no state in the table for k, we apply the same process, but consider i' to be 0. In both cases, we put the generated states in the table for $k + 1$, if they are not dominated by a state already present. The removal of items from a single infeasible state is also demonstrated in Figure 4.

This algorithm will find a solution for a certain size $w > b - a_{\max}$ if it exists, since

for each such w a solution can be derived from $\hat{x}$ by adding and removing items in the correct order. The exact argument is presented in Section C.2.

Now when we want to calculate the table for $k = |I|$ by calculating all previous tables, a simple observation tells us that, since there are at most $O(|I|)$ possible values and we need at most $O(a_{\max}|I|)$ time to calculate a table, we need $O(a_{\max}|I|^2)$ to calculate the table for $k = |I|$. However, this analysis is not tight. While it is true that we may perform $O(|I|)$ steps when removing items from a single state with size $w' > b$, all future operations on states of this size w', it is not necessary to consider the removal of items that have been considered for removal before. If item i' was remove previously from our current state, the state at $w' - a_{i'}$ will have an i at least $i' - 1$. Therefore, we only have to remove items with indices $> i'$ from our current state, since the removal of items with indices $\leq i'$ will result in states that are dominated anyway. So, over the course where k takes on all values from $\hat{j}$ to $|I|$, the total number of removals from states with size w' is bounded by $|I|$. Now when we separately analyze the number of states examined for all values $w' > b$ and the number of item removals performed for states $w' > b$, we can see that there are $O(a_{\max}|I|)$ states and $O(a_{\max}|I|)$ removals. Since all additions could be done in constant time and we examine $O(a_{\max}|I|)$ states with $w \leq b$, the true time bound for this algorithm is $O(a_{\max}|I|)$.

The full process of constructing the table for $k + 1$ from a table k is presented in Algorithm 3. When we take a look at the algorithm, we see a single for loop over w' for the copy operations, then a single for loop over w' for the add operations and finally a double nested for loop over w' and j. Since the body of the first two for loops is constant, copying and addition take $O(a_{\max})$ time. When we consider removals, the worst case gives us a situation where we must remove all items for each value of w', so the removals may take up $O(a_{\max}|I|)$ time. Thus, calculating a single table takes $O(a_{\max}|I|)$ time.

This approach can be extended to knapsack problems, by taking the current profit of a solution into account. By using a gap Γ between a upper and lower bound on the solution value, the algorithm can solve Knapsack problem in $O(|I|a_{\max}\Gamma)$ time. The exact way to do this is presented in the work by Pisinger [Pis99].

5.2.3. Dynamic Programming for PCKP with Trees

Now consider the Precedence Constrained Knapsack Problem where the precedence graph has a tree structure. We will assume this tree structure to be an out-tree (i.e. the root has no precedence constraint, all children of the root can be added after the root is added, etc.). We will see this structure when we discuss the Surrogate Relaxation in Section 5.4.5, but the algorithm can also be adapted to a variant for an in-tree. For more detailed information on these algorithms, see [JN83].

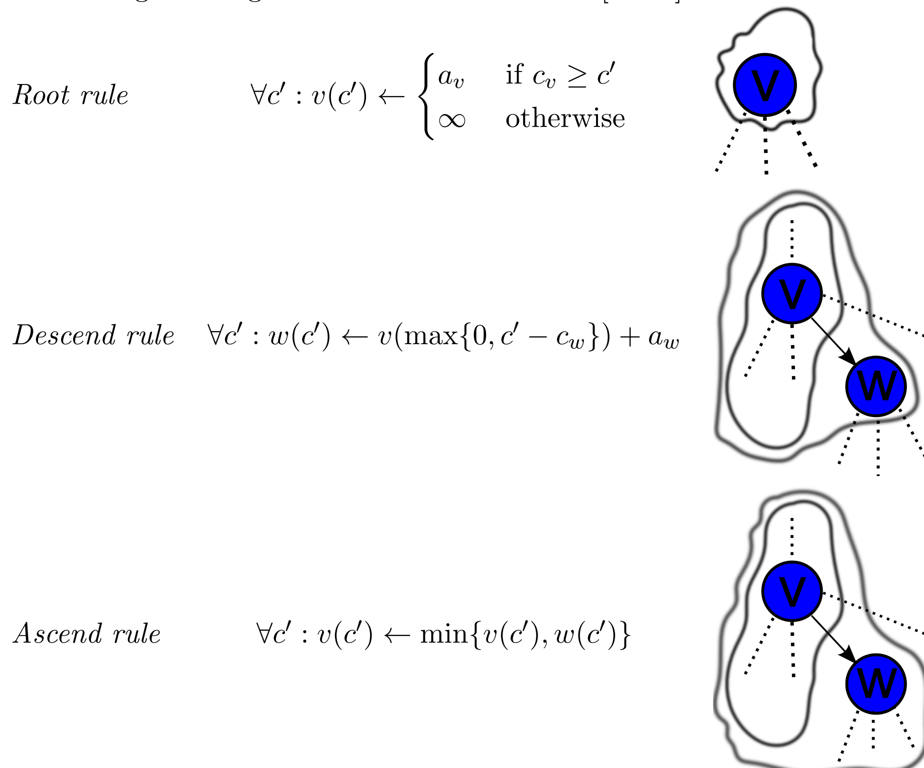
Let us number the tree by performing a depth first preorder traversal. This implies we number a node when we visit it. We first visit the root of the tree, which will get number 1. We then visit the subtree induced by the first child of the root, so all nodes under the first child are numbered. Then we visit the subtree induced by the second child of the root, etc. Since each node in the tree represents an item, we have $|I|$ nodes. Let us introduce the function $s(i)$ that gives the number of children node i has (i.e. if i

Algorithm 3 Calculate table $k+1$ from table k

Input: Table A
Output: Table B
Initialize Table B
for $w' \leftarrow b - a_{\max}$ to $b + a_{\max}$ **do**
  $B[w'] \leftarrow A[w']$
end for
for $w' \leftarrow b - a_{\max}$ to b **do**
  if $A[w']$ contains an entry **then**
    $(k', i', w') \leftarrow A[w']$
    $(k+1, i, w' + a_{k+1}) \leftarrow B[w' + a_{k+1}]$
    if $i' > i$ **then**
      $B[w' + a_{k+1}] \leftarrow (k+1, i', w' + a_{k+1})$
    end if
  end if
end for
for $w' \leftarrow b + a_{\max}$ to $b + 1$ **do**
  if $B[w']$ contains an entry **then**
    if $A[w']$ contains an entry **then**
      $(k, i', w') \leftarrow A[w']$
    else
      $i' \leftarrow 0$
    end if
    $(k+1, i, w') \leftarrow B[w']$
    if $i' < i$ **then**
      for $j \leftarrow i$ to $i' + 1$ **do**
        $(k+1, i'', w' - a_j) \leftarrow B[w' - a_j]$
        if $i'' < j$ **then**
          $B[w' - a_j] \leftarrow (k+1, j, w' - a_j)$
        end if
      end for
    end if
  end if
end for

has three children, $s(i) = 3$).

During our traversal of the tree, a state table is stored in each node on the path from the current node to the root. A state (c', u') implies that the minimum size for a knapsack with at least profit c' is u'. The notation $v(c') = u'$ implies we have a state (c', u') stored in the table of a node v. If we traverse an arc (v, w) in the tree, we will apply a rule to update the table in one of the nodes: if we traverse the arc from v to w we apply the descend rule, while we apply the ascend rule if we traverse the arc from w to v. After we have traversed the entire tree and are back in the root, the root node contains the table with the best values for the complete tree. The rules are derived from the original Dynamic Programming recurrence in the work of [JN83].



The *Root rule* is easy to understand: we must add the root node to the tree (otherwise we have an empty knapsack). The *Descent rule* is a generalization of this principle: if we descend the tree, we must add the node to our solution (otherwise we can't add it's children). The *Backtrack rule* implies choice: when we ascend to a node that has multiple children, it can choose between the paths of these children. Before ascending an arc (v, w), node v contains the table for all children of v up to w and node w contains the table with all children with w a mandatory node. The ascend rule states that we from the point of view of node v, we must only consider w mandatory if it improves our solution.

Now let us consider the running time of the algorithm. Suppose we have some upper-bound U on the value of the solution. We can limit the size of our tables to U. During the depth first preorder traversal, each node will be reached once by descending to it,

and once the results of the entire subtree of that node and its successors are calculated, a single ascend operation to its parent will be performed. This means each node except the root will perform exactly two operations, that both take $O(U)$ time to calculate. Since each node takes $2U$ time to calculate, except for the root, the running time of the algorithm will be $O(2U(|I|-1)) = O(U|I|)$. Since the algorithm needs to store a table in memory for each node in the current path of the traversal, we can use a bound on the path length (i.e. the depth of the tree) d for the bound on the memory: $O(dU)$ when we are considering a decision variant of the problem.

The algorithm can be transformed to an algorithm with tables $v(a') = c'$, which gives a running time of $O(b|I|)$ and a memory use of $O(d|I|)$.

5.2.4. Dynamic Programming for kKP

Now consider the kKP, the variant of Knapsack where we must find a knapsack of at most k items. This problem is very close to the original Knapsack problem. For the original problem, our main concern was the capacity of the solution. This remains a concern, but the cardinality of the solution is a new concern. Still, the principle of the optimal substructure that was used to construct a Dynamic Programming algorithm for the KP can be extended to take the cardinality of the solutions into account. This can be achieved by adding the cardinality as a parameter to the Dynamic Programming recurrence, as shown by [CKPP98]. Now when we add an item to a previous solution, the cardinality of that solution is increased by one (and the total size is increased by the size of the item). We will define a recurrence $B(i, k', w)$, which gives the profit of the best possible knapsack that contains a subset of items that have indices j such that $j \leq i$, contains exactly k' items and has exact weight w.

$$
\begin{array}{lcll}
B(i,0,0) & = & 0 & \\
B(i,0,w) & = & -\infty & w \neq 0 \\
B(i,k',0) & = & -\infty & k' \neq 0 \\
B(i,k',w) & = & \max \left\{ \begin{array}{l} B(i-1,k',w) \\ B(i-1,k'-1,w-a_i)+c_i \end{array} \right. &
\end{array}
$$

The table can be calculated in the same fashion as the Dynamic Program from section 5.2.1, but it has k as an extra dimension to the table. This yields a running time of $O(|I|bk)$ time and $O(bk)$ space. Of course it is also possible to extend the Balanced Dynamic Programming recurrence from section 5.2.2, to obtain a faster algorithm.

5.3. Dynamic Programming for Size Robust Knapsack Problems

5.3.1. Robustness and the Knapsack Problem

Now that we have discussed the basic Knapsack Problem, we can discuss some robustness variants for the problem. While robustness problems with scenarios in which the weights of the items differ for each scenario or jobs take more or less time than expected have been discussed by [BKK11], we will concentrate on scenarios where our capacity constraint differs. We introduce the scenario set S with a capacity b_s for each scenario $s \in S$. Since

we may want to consider expected cost objectives, we will also define a probability p_s for each scenario in such a way that our probabilities sum up to 1.

In this process, we will consider the initial and recovery solution separately. While both solutions consist some item-set, we want an initial item-set and a separate recovery item-set for each scenario. The item-set for each scenario should be recovered from the initial set, by some means of recovery. The capacity for the initial knapsack has a special symbol, b_0. We also reserve the index 0 for the initial "scenario", which can also be seen as a special case of a recovery scenario that doesn't need any recovery from a feasible initial solution.

Now that we have our scenarios, we will discuss different ways of recovery. We will assume that b_0 is equal to or larger than all scenario capacities. Our initial solution may not exceed the capacity b_0. We will consider means of recovery where we are allowed to remove items, or means of recovery where we are allowed to swap items for other items. First we will consider a simple recovery procedure where recovery is done in a greedy way, according to a given order, like value or weight. This implies that recovery is done by removing the cheapest item first, or the largest item first, until the solution becomes feasible. Such an extension of the problem is called the Size Robust Knapsack Problem *with Greedy Recovery* (by cheapest item, or by largest item, by lowest ratio, etc). Since we are allowed to remove all items that are currently in the knapsack, a solution to such a problem will always be feasible (since a solution without items will never violate the capacity constraint). Such a recovery scheme isn't interesting with respect to the strict interpretation of recoverable robustness, since the feasibility is trivial. However, it can be interesting to find a solution that has good expected profits.

If we limit the number of items we are allowed to throw away, feasibility becomes an issue. It is possible that putting too many items in our knapsack will make it impossible to throw enough items away to make our knapsack feasible. If we are only interested in finding a feasible solution (and thus look at the problem from a Recoverable Robustness perspective), we only need to consider our main capacity b_0 and the lowest capacity for all values of b_s. If we can recover to the lowest value of b_s, we also know that we can recover to larger capacities, because the recovery for the lowest value is also feasible for all larger values. If we place a limit on the number of items that we can throw away during recovery, we call this *cardinality constrained recovery.* It is also possible to combine this recovery method with an objective - the expected costs are a likely candidate.

Now suppose we are not allowed to throw any item away, but we may swap them for other items. This gives us another recovery procedure. Feasibility becomes an issue, because in an instance where we select all items, we cannot recover at all, since we can't swap any items. Of course, the number of swaps can be constrained, making feasibility an even greater issue. When we recover using swaps, we call the problem the Size Robust Knapsack Problem *with Recovery by Swapping.* Like with the cardinality constrained recovery, we can look at these problems from a Recoverable Robustness perspective, since feasibility is not trivial. When we do this we also can limit ourselves to only the main capacity b_0 and the minimal capacity for all b_s values, using the same argument again. Of course, it is also possible to consider this problem with an objective function - again the expected costs are a likely choice for an objective function.

The most basic form of recovery, is recovery where we can throw away items, but have no limitations on what items to throw away and even have no predefined method to do this. In such a case, finding the best way to remove items from a certain item-set is a Knapsack Problem in itself - we need to choose a set of items from the main item-set in such a fashion that we retain the highest possible profit, but do not exceed our capacity of the scenario. Since feasibility is trivial in this case, this problem is more on the two-phase stochastic programming side of the spectrum.

Since we introduced a lot of concepts for recovery, we will give a short summary of the terminology. We will also present abbreviations that can be used to express all variants of the Size Robust Knapsack Problem (in short: RKP) and the Robust Subset Sum Problem (in short: RSSP). When we describe a problem, we will use the scheme: *"objective"-problem-"recovery constraint"*.

Objectives

Feasibility Objective (F-) We are only concerned with finding an initial solution with maximum profit that can be made feasible for each scenario. This implies we view the problem from a Recoverable Robustness perspective.

Worst Case Objective (W-) We are concerned with finding an initial solution such that the lowest maximum profit for a single scenario is maximized. In other words, we want to perform as good as possible in the worst case.

Expected Profit If no prefix is given, we want to maximize the expected profit. The expected profit is given by the sum of the profits for each scenario times the probability of the scenario.

Recovery Constraints

Recovery by Removal (-R) We are only allowed to throw away items from our initial solution.

Recovery by Swapping (-S) We are only allowed to swap items that we used in the initial solution for items that were not used in the initial solution to create solutions for the scenarios.

Cardinality Constrained Recovery (-C) We have a limited number k of recovery steps that we may do to derive a feasible solution for our scenario from the initial solution.

Greedy Recovery (-G) We have some rule on the order in which we should throw our items away. Likely candidates are largest weight (-G$a_{\max}$), smallest profit (-G$c_{\min}$) or smallest ratio (-G$(\frac{c}{a})_{\min}$), but other rules are possible.

When we create an abbreviation for a certain type of Size Robust Knapsack Problem, the objective type is a prefix for RKP, while the recovery types are postfixes for the RKP. For example, the Feasibility Size Robust Knapsack Problem with Cardinality Constrained Recovery by Swapping is abbreviated to F-RKP-SC, while the Size Robust Knapsack

Problem with Greedy Recovery by Removal on the Smallest Ratio is abbreviated to RKP-RG$(\frac{c}{a})_{\min}$.

5.3.2. Dynamic Programming for F-KP-RC

Let us consider the Feasibility Knapsack Problem with Cardinality Constrained Recovery by Removal. A simple idea to solve the problem is to consider the items ordered from small to large. In this way, we know that we cannot add more than $k-1$ items to a possible solution from the moment the summed weight of the item-set becomes larger than $b_{\min}$. We will use this principle to show that the F-KP-RC is as least as hard as the kKP problem, since the kKP problem is a special case of the $F-KP-RC$.

We will now consider a couple recoverable robustness knapsack problems and present some techniques to solve them.

Theorem 5.1. *The kKP is a special case of F-KP-RC.*

Proof. Suppose we have an instance of kKP with capacity b and the cardinality of the knapsack constrained to be at most k. Let us create an instance of the F-KP-RC problem, with $b_{\max} = b$ and $b_{\min} = 0$, and a cardinality constraint on the recovery of k. The optimal solution to this F-KP-RC problem can contain at most k items, since recovery to the empty set should be possible by throwing away at most k items. If the solution contains more than k items, recovery is not feasible. □

While the F-KP-RC is a generalization of the kKP, we can adapt the algorithm presented in Section 5.2.4 to make it work for the F-KP-RC. While the algorithm for kKP makes no assumption about the order of the items, we will assume items are ordered according to their sizes, such that the last item always has greatest size. This way, we can easily find our maximum recoverable size: we just remove the last k items. If we combine this idea with the observation that after we exceed capacity $b_{\min}$, we can only add $k-1$ additional items to our solution. Let us redefine the meaning of k' in the state (i, k', w) of the Dynamic Program for the kKP in such a fashion that it represents the amount of items that make our capacity exceed $b_{\min}$. Since the order implies that by doing this, we only count the largest items in the solution under consideration, it is not very difficult to see that this yields a correct algorithm.

Modifying the algorithm, i still represents the item under consideration for addition and w represents the size of the current solution in the state (i, k', w).

$$
\begin{aligned}
C(i,0,0) &= 0 && \\
C(i,0,w) &= -\infty && \text{for } w > 0 \\
C(i,k',0) &= -\infty && \text{for } k' > 0 \\
C(i,k',w) &= \max \begin{cases} C(i-1,k',w-a_i)+c_i \\ C(i-1,k',w) \end{cases} && \text{if } w \leq b_{\min} \\
C(i,k',w) &= \max \begin{cases} C(i-1,k'-1,w-a_i)+c_i \\ C(i-1,k',w) \end{cases} && \text{if } w > b_{\min}
\end{aligned}
$$

5.3.3. Dynamic Programming for RKP-S

When we consider recovery with swapping without a limit on the number of swaps, the problem can be solved by Dynamic Programming. The idea is that when the allowed number of swaps is at least $\lceil\frac{|I|}{2}\rceil$, we can recover from one set of items to another set of items as long as the two sets have the same cardinality, since they can be transformed into each other by swapping an item that is not in the target set for an item that is. Using this knowledge we can use the Dynamic Programming recurrence B for the kKP from Section 5.2.4. Let us recall $B(i,k,w)$, which gives the value of the best possible knapsack with size w, cardinality k, containing items with indices up to i. We use Algorithm 4 to find a solution to RKP-S.

$$
\begin{array}{lcll}
D(i,0,0) & = & 0 & \\
D(i,0,w) & = & -\infty & w \neq 0 \\
D(i,k',0) & = & -\infty & k' \neq 0 \\
D(i,k',w) & = & \max \left\{ \begin{array}{l} D(i-1,k',w) \\ D(i-1,k'-1,w-a_i)+c_i \end{array} \right. &
\end{array}
$$

Algorithm 4 Dynamic Programming Algorithm for RKP-S

Calculate table $D(|I|,k,w)$
Initialize $c^* \leftarrow 0$
for $1 \leq k \leq |I|$ **do**
  Initialize $c_k \leftarrow 0$
  for $s \in S$ **do**
    $c_s \leftarrow p_s \max\limits_{0 \leq w \leq b_s} C(|I|,k,w)$
    $c_k \leftarrow c_k + c_s$
  end for
  $c^* \leftarrow \max\{c^*, c_k\}$
end for
return c^*

To prove the correctness of Algorithm 4, we prove a theorem that shows the relation between swapping operations and the cardinality of item-sets.

Theorem 5.2. *Consider an item-set I and two item-sets $I' \subseteq I$ and $I'' \subseteq I$. If we are only allowed to use swap operations, I' can only be transformed into I'' if and only if the cardinality of I' is equal to the cardinality of I'' (i.e. $|I'| = |I''|$).*

Proof. If $|I'| = |I''|$ we can transform I' into I'' by using the following procedure

while $I' \neq I''$ **do**
  Choose i_1 such that $i_1 \in I', i_1 \notin I''$
  Choose i_2 such that $i_2 \notin I', i_2 \in I''$
  $I' \leftarrow$ swap i_1 for i_2 in I'

end while

If $|I'| \neq |I''|$ we can never transform I' into I'', since the cardinality of I' will not change when we use a swap operation. Since two identical item-sets have the same cardinality, I' will never become I'' by swapping only. □

Since recurrence D finds the best solutions for each cardinality and the proof implies that swapping restricts recoverability to item-sets of the same cardinality, we have shown that Algorithm 4 finds an optimal solution. Now let us consider the running time of Algorithm 4. Calculating the full table $D(|I|, k, w)$ takes $O(|I|^2 b)$ time. In the algorithm itself, the outer loop takes $O(|I|)$ time, the inner loop takes $O(|S|)$ time and the max over $0 \leq w \leq b_s$ takes $O(b)$ time, assuming the values of C are already calculated. This gives a total running time of $O(|I|^2 b + |I||S|b)$.

5.3.4. Dynamic Programming for RKP-RG

When we consider these greedy recovery methods, we can solve these problems with the general Dynamic Programming approach we used for the normal KP, if we sort the items according to the inverted ordering of our greedy recovery. If our greedy recovery states we should recover by removing the smallest item first, we should add the greater item i before j if $a_i > a_j$. In general, for any two items i and j in our knapsack such that i was added before j, item j will be removed before item i will be removed by our greedy recovery algorithm. This has the advantage that in any case where we consider to add an item in our Dynamic Programming algorithm, we know that the new item will always be removed first when recovery is done. We will define a recurrence $E(i, w)$, that gives the value of the best possible solution to the RKP-RG considering items with indices j such that $j \leq i$ and weight w for the initial solution. We will use the notation $[b_s \geq w]$, which has value 0 when $b_s < w$ and value 1 when $b_s \geq w$ indeed holds.

$$
\begin{aligned}
E(i, 0) &= 0 \\
E(0, w) &= -\infty \\
E(i, w) &= \max \begin{cases} E(i-1, w) \\ E(i-1, w-a_i) + \sum_{\forall s \in S} [b_s \geq w] p_s c_i \end{cases}
\end{aligned}
$$

When compared to the original recurrence, we can see that c_i is replaced by $\sum_{\forall s \in S} [b_s \geq b] p_s c_i$. In other words, we only add $p_s c_i$ for the scenarios where the capacity constraint is not exceeded. The correctness can be proven from the fact that recovery is done in the inverse order in which items are added.

5.3.5. Dynamic Programming for RSSP-R and RKP-R

In this section we will show that for a fixed number of scenarios (e.g. $|S| = 2$), the problem remains solvable in pseudo-polynomial time. To achieve this we will first take a look at the Subset Sum Problem with Exact Recovery by Removal problem (in short: E-RSSP-R) with two scenarios. We will consider b_0 as scenario - so we have a scenario where we don't have to recover and a scenario where we may have to throw some items

away. This gives us a decision problem: is there a solution to the problem where we can find a knapsack of exactly size b_0, such that this knapsack contains a subset of exactly size $b_0 - b_1$. This problem can be reformulated as a partition in three parts problem: can item-set $|I|$ be partitioned into three disjoint sets, with set 1 exactly size b_1, set 2 exactly size $b_0 - b_1$ and set 3 the remainder of the items?

Lemma 5.3. *The e-RSSP-R with two scenarios is equivalent to partition in three parts.*

Proof.

1. If the partition in three parts problem has a solution, the e-RSSP-R has a solution. Suppose we have a solution for the partition problem. From this solution we will derive three disjoint sets I_1, I_2 and I_3, with $\sum_{i \in I_1} a_i = b_1$ and $\sum_{i \in I_2} a_i = b_0 - b_1$. We will take the union of I_1 and I_2 as the initial solution (which is feasible, since $b_1 + b_0 - b_1 = b_0$. We will take I_1 as the solution for scenario 1, which is feasible by definition (it is a subset of a union of itself and another set and its size is b_1).

2. If the e-RSSP-R problem has a solution, we can derive a solution for the partition in three parts problem. Suppose we have some set I_0 as a solution for the main scenario, and I_1 as the knapsack we use for scenario 1. Suppose we need to find a partition (I'_1, I'_2, I'_3), such that I'_1 must have size b_1, I'_2 must have size $b_0 - b_1$ and I'_3 contain the remainder of the items. We will use I_1 for I'_1, which is feasible by definition. Since I_1 must be a subset of I_0, we can use $I_0 - I_1$ for I'_2, and use all unused items for I'_3.

□

Knowing that the 2-scenario problem can be solved using partition in three parts, we take a look at a Dynamic Programming algorithm that is capable to solve this problem, which is just a multi-dimensional variant of the Bellman recurrence, as shown in [Bel57].

Now let us apply this observation to derive a Dynamic Programming algorithm for the $RKP - R$. If we have a new item to consider, we must put it in the main item-set and any subset of the lower scenario item-sets, or don't put it in any item-set at all. Let us introduce the state variable $F_1(i, w_0, w_1)$, which gives the best value for a main knapsack and it's recovery knapsack, such that the recovery knapsack is a subset of the main knapsack, where the recovery knapsack has size w_0 and the main knapsack has size w_1. Now let's consider the situation where we want to solve the problem for three scenarios. We can extend the recurrence in the following way:

$$
\begin{array}{lcl}
F_2(i, 0, 0) & = & 0 \\
F_2(0, w_0, w_1) & = & -\infty \text{ if } w_0 \neq 0 \vee w_1 \neq 0 \\
F_2(i, w_0, w_1) & = & \max \begin{cases} F_2(i-1, w_0, w_1) \\ F_2(i-1, w_0 - a_i, w_1) + p_0 c_i \\ F_2(i-1, w_0 - a_i, w_1 - a_i) + (p_0 + p_1) c_i \end{cases}
\end{array}
$$

It is possible to adapt this idea to derive a recurrence $F_3(i, w_0, w_1)$ that works when we have 3 scenarios. The state variable $F_3(i, w_0, w_1, w_2)$ gives the best value for a main

knapsack of exactly weight w_0 and two recovery knapsack of exactly weight w_1 and w_2, such that both the recovery knapsacks are subsets of the main knapsack.

$$
\begin{array}{lcl}
F_3(i,0,0,0) & = & 0 \\
F_3(0,w_0,w_1,w_2) & = & -\infty \text{ if } \exists s \in S : w_s = 0 \\
F_3(i,w_0,w_1,w_2) & = & \max \begin{cases} F_3(i-1,w_0,w_1,w_2) \\ F_3(i-1,w_0-a_i,w_1-a_i,w_2-a_i) + (p_0+p_1+p_2)c_i \\ F_3(i-1,w_0-a_i,w_1-a_i,w_2) + (p_0+p_1)c_i \\ F_3(i-1,w_0-a_i,w_1,w_2-a_i) + (p_0+p_2)c_i \\ F_3(i-1,w_0-a_i,w_1,w_2) + p_0c_i \end{cases}
\end{array}
$$

Of course we can extend this approach to a situation where we have $|S|$ scenarios:

$$
\begin{array}{lcl}
F_{|S|}(i,0,0,\dots,0) & = & 0 \\
F_{|S|}(0,w_0,w_1,\dots,w_s) & = & -\infty \text{ if } \exists s \in S : w_s = 0 \\
F_{|S|}(i,w_0,w_1,\dots,w_s) & = & \max \begin{cases} & F_{|S|}(i-1,w_0,w_1,\dots,w_s) \\ \max\limits_{\sigma \in \varsigma} & F_{|S|}(i-1,w_0-a_i,\dots,w'_s-[s' \in \sigma]a_i,\dots) \\ & +\sum_{s\in\sigma} p_s c_i \end{cases}
\end{array}
$$

$$
\begin{array}{ccc}
 & \text{where} & \\
\varsigma & = & 2^S (\text{i.e. the powerset of } S) \\
[s \in \sigma] & = & \begin{cases} 1 \text{ if } s \in \sigma \\ 0 \text{ if } s \notin \sigma \end{cases}
\end{array}
$$

The powerset is the set that contains all subsets of a given set, including the empty set. Proving the correctness of this algorithm can be done in a similar way as used in Section C.1.

5.3.6. Branch and Bound for RKP-R

Let us consider Branch and Bound to solve the RKP-R. If we consider the ILP-formulation, we have many a lot of variables and we would consider to branch on each item for each scenario. Since Branch and Bound takes more time when it has more branching options, we will consider a way to branch at most once on each item. We will introduce a tree-based data structure to represents a knapsack of items and keeps track of the optimal recovery solutions. We use the ideas from the original DP for the KP (discussed in Section 5.2.1), to keep track of a recovery table. When we add an item to a knapsack, we calculate a new recovery table, based on the old recovery table and the new item. This allows us to efficiently calculate the exact profit, recovery included, of a knapsack. We introduce the RecoveryKnapsack data structure in Algorithm 5.

Now if we use the RecoveryKnapsack we can branch on the addition or omission of items in the initial knapsack and we will always get the exact recovery value of a current knapsack. In a typical Branch and Bound tree for the KP, each depth-level of the tree is associated with a particular item and each node has two possible branches: one which

Algorithm 5 RecoveryKnapsack Data Structure

```
Object variable: recoveryTable ← new Table[b_0 − b_|S|] initialize with ∞
Object variable: item ← NULL
Object variable: size ← 0
Object variable: profit ← 0
Object variable: parent ← NULL

Method expand(i) {Creates a new KP node and calculates recovery}
Variable result ← new RecoveryKnapsack()
result.item ← i
result.size ← this.size + a_i
result.profit ← this.profit + c_i
result.parent ← this
for 1 ≤ w ≤ b_0 − b_|S| do
  if a_i ≤ w then
    result.recoveryTable[w] ← min { this.recoveryTable[w]
                                    c_i
  else
    result.recoveryTable[w] ← min { this.recoveryTable[w]
                                    this.recoveryTable[w − a_i] + c_i
  end if
end for
return result

Method getObjectiveValue() {Calculates the objective value for this node}
Variable result ← this.profit
for s ∈ S, s ≠ 0 do
  if this.size − b_s > 0 then
    result ← result − p_s · this.recoveryTable[this.size − b_s] {We assume Σ_{s∈S} p_s = 1}
  end if
end for
return result

Method contains(i) {Checks whether the KP this nodes represents contains i}
Variable current ← this
while current ≠ NULL do
  if current.item = i then
    return True
  end if
  current ← current.parent
end while
return False
```

includes the item at its depth and one which excludes the item. Now there are a lot of possibilities to order the items, which will influence the depth at which an item is considered. We may consider orderings by smallest or greatest size, smallest or greatest profit or smallest or greatest ratio. Combined with the order in which we omit or include items, we have 12 different possible branching strategies in a typical Branch and Bound algorithm for KP. However, a good Branch and Bound algorithm benefits from having a good upper-bound algorithm as well, since this allows us to bound the tree better. The next section will consider methods to calculate such upper and lower-bounds.

5.4. Upper and Lower-bounds

5.4.1. Iterative DP Lower-bound for RKP-R

Let's reconsider the basic DP for knapsack. In this Dynamic Program, new knapsacks are generated by adding a single item to them (by incrementing the item index by one each time). Now let's say we focus on finding a initial knapsack solution for the RKP-R. In Algorithm 5, we presented a data structure to efficiently calculate the optimal recovery value for each scenario when only adding items sequentially.

Let us combine this data structure with the basic DP for knapsack to find an initial item-set for the RKP-R, while taking recovery into account. Since the value is not directly dependent on which item is added (because of the recovery), we will not use solution values in the recurrence, but create a recurrence that contains an instance of our data structure instead. In this recurrence $G(i, w)$, i represents the index of an item while w represents the size of the knapsack we are currently considering.

$$
\begin{aligned}
G(0,0) &= \textit{new}\ \text{RecoveryKnapsack()} \\
G(0,w) &= \text{empty, for } w > 0 \\
G(i,w) &= \text{best of} \begin{cases} G(i-1,w) \\ G(i-1,w-a_i).\text{expand}(i) & \text{if } G(i-1,w-a_i) \text{ nonempty} \end{cases}
\end{aligned}
$$

While this approach seems to work good in practice, it gives no guarantee in finding the optimal solution. The major problem is that the domination criterium is not fully valid in this case: it is possible that adding an item to a non-optimal set for a certain size w' yields add a better solution than adding that same item to the optimal solution for w'. However, if all the items in the optimal initial solution are added before adding any other items, this approach will find the optimal solution. Because of this, one might believe we may find the optimal solution by repeating this scheme $|I|$ times, since in this way, each possible permutation of the items is contained as a sub-sequence in the DP. However, this approach is still not valid, since it is possible that each possible subset of the optimal solution is dominated by a subset that can't lead to the optimal solution. An example of this is shown in Table 2. In this case, the optimal solution doesn't contain item 3, but if we remove any item from the optimal solution, the remaining subset is dominated by a set that contains 3.

Still, based on experiments conducted in Chapter 6, this approach can be applied with some success as an lower-bound algorithm. Since the order of the items is important for

Items:

i	1	2	3	4
a_i	8	4	6	2
c_i	290	170	241	70

Scenarios:

s	0	1
b_s	15	13
p_s	0.5	0.5

Feasible Sets:

Items	Size	Profit	Scenarios	Recovery Value
$\{1, 2\}$	12	460	$\{0, 1\}$	460
$\{1, 3\}$	14	531	$\{0\}$	410.5
$\{1, 4\}$	10	360	$\{0, 1\}$	360
$\{2, 3\}$	10	411	$\{0, 1\}$	411
$\{2, 4\}$	6	240	$\{0, 1\}$	240
$\{3, 4\}$	8	311	$\{0, 1\}$	311
$\{1, 2, 4\}$	14	530	$\{0\}$	486
$\{2, 3, 4\}$	12	481	$\{0, 1\}$	481

Subsets from the optimal set are dominated

Set	Dominated by
$\{1, 2\}$	$\{2, 3, 4\}$
$\{1, 4\}$	$\{2, 3\}$
$\{2, 4\}$	$\{3\}$

Table 2: Example where the Iterative DP can go wrong

the solution we will find, we may consider different strategies for ordering our items. The whole algorithmic scheme for this Iterative Dynamic Programming approach is presented in Algorithm 6. Since the order of the items is important, we may choose different strategies to reorder our items after each iteration. We may consider orderings based on size, profit or ratio. It is also possible to reverse the order of the items after each iteration. Another idea is to randomize the order of the items after each iteration. These four options are all explored in Chapter 6.

5.4.2. Local Search Lower-bounds

By using Local Search, we can search for solutions without a real guarantee on the optimality of the solution. Such a solution is a lower bound for the solution value of the problem. One of the important aspects of Local Search is the way you define your solution space. When we consider robustness variants of the Knapsack Problem, a solution should at least contain which items are chosen initially. For the recovery solutions, we have two choices: we explicitly keep track of all recovery solutions (and define operations on the recovery) or we focus on the initial solution and take recovery implicitly into account by using the objective value of the current solution, including recovery.

When we take the implicit approach, we need to calculate the recovery on the fly. A possible way to do this is by using a recovery variant of the basic knapsack Dynamic Program from section 5.2.1. This way a solution consist only of the initial choices, since the additional properties are implied by solving the recovery problem for the initial solution.

Depending on whether we define our recovery solution explicitly or implicitly, we also have different choices for our neighborhoods. If we have an explicit solution space, we need operators that can add and remove items from the initial part of the solution, but also from and to each of the recovery solutions. In addition to that, our operators need to be constrained by the recovery constraints, yielding different operations for cases where we recover by removal and cases where we recover by swapping.

Considering the solution space, it is plain to see that the solution space with implicit solutions is much smaller than the solution space with explicit solutions. For example, if we consider recovery by removal, we can have a solution where we have a lot of items in the initial item-set, but no items in any of the scenario items, while there is room for items in these sets. Such a solution can be represented by using the explicit variant of the solution space, but not by the implicit version, since the recovery is not optimal in this case. The implicit variant will calculate the optimal recovery and thus give a better solution for the same initial item-set.

However, the drawback of the implicit variant is the fact that it needs to perform the recovery algorithm on each new solution. Since the recovery algorithm needs to solve a knapsack problem, this can take up a lot of time. In cases where we add an item, we can calculate a single column, if we keep the previous Dynamic Programming tables in memory. When an item is removed, it is possible a lot of work needs to be done. Calculating the exact value of an explicit solution is much easier, since the exact value of the solution only depends on each item-set.

Algorithm 6 Iterative DP for RKP-R

```
Initialize Global table ← new Table [b]
table[0] ← newRecoveryKnapsack() {See Algorithm 5}

Procedure IterativeDP
Initialize: improve ← True
Initialize: iteration ← 0
Initialize: lastBest
while improve ∧ iteration < |I| do
   improve ← False
   Reorder(I, iteration) {Reorders the items according to some procedure}
   doDP() {Executes the doDP() procedure from below}
   best ←findBest(table) {Retrieves the best solution currently in table}
   if best > lastBest then
      improve ← True
      lastBest ← best
   end if
   iteration ← iteration + 1
end while
return lastBest

Procedure doDP
for i ∈ I do
   Initialize newTable ← new Table [b] {Initialize a new table}
   for 0 ≤ w ≤ b do
      if w − a_i < 0 ∨ i ∈ table[w − a_i].items() then
         newTable[w] ← table[w] {If w too small or i already used, keep the old solution}
      else
         if a ≥ 0 ∧ ¬table[w − a_i].contains(i) then
            newTable[w] ← table[w − a_i].expand(i) {Add i and calculate recovery}
            c_1 ← table[w].getObjectiveValue()
            c_2 ← newTable[w].getObjectiveValue()
            if c_1 > c_2 then
               newTable[w] ← table[w] {Retain the best option for size w}
            end if
         end if
      end if
   end for
   table ← newTable {Store the new table at the location of the old one}
end for
return
```

Besides implicit and explicit solutions, we can also explore balanced variants of the solutions, based on the ideas presented in section 5.2.2. With this approach we allow invalid solutions. We may only add items if the capacity constraints are not violated and can only remove items if they are violated. The difficulty of such an approach lies in the fact that we have two kinds of solutions - feasible and infeasible solutions. Since the infeasible solutions are usually very close to feasible solutions, it would be bad practice to give them a negative solution value, but since there are infeasible they should have a lower value than the feasible solutions. If this isn't the case, the Hillclimbing algorithm will refuse to decrease the solution value of such a solution and therefore refuses to make the solution feasible again.

The nice part about using the balanced approach is that besides giving a lower-bound (if we keep track of the feasible solution with the highest value we have seen), it also gives an upper-bound by keeping track of the infeasible solution with the lowest value we have seen.

5.4.3. LP-Relaxation Upper-bound

Now let us consider an upper-bound. One of the simplest upper bound algorithms for the Knapsack Problem is the Linear Programming relaxation. In general, the Linear Programming relaxation removes the constraint that the variables should have integer values in the solution. In the Knapsack Problem, this implies that the LP-relaxation allows us to take an arbitrary part of each item, instead of the item as a whole. This approach is quite straightforward and discussed in both textbooks on the Knapsack Problem, [MST90] and [KPP04].

Let us consider our items in a descending order by ratio, so item $\frac{c_i}{a_i} > \frac{c_{i+1}}{a_{i+1}}$. Since we can take arbitrary parts of each item, we can fill our knapsack to a point where the sum of the weights is equal to the capacity constraint. Now we can swap some part for any item in the knapsack for a part of equal size that is not currently in the knapsack. If we swap a part of item i for a part of equal size of item j, our solution value will improve if and only if $\frac{c_j}{a_j} > \frac{c_i}{a_i}$. Now suppose we add items to our knapsack in order of descending ratio, until an item $\bar{i}$ doesn't fit. We will take the part of $\bar{i}$ that fills up the knapsack. Now assume we can improve our solution by swapping. We will find that for any part of any item i in the knapsack, all parts of all items j that have $\frac{c_i}{a_i} > \frac{c_j}{a_j}$ are already in the knapsack and can't be added a second time. Since it is impossible to improve the solution, it must be optimal.

Now suppose we have the RKP-R and repeat the same process to find initial and recovery solutions for each of the scenarios. Since the ordering by ratio is the same for each item-set, the order in which we add the items will be the same. This implies that, if the initial capacity constraint gives the greatest capacity, no recovery item-set for any scenarios will have a larger part of any item in it than the initial item-set has. Thus, applying this approach gives a feasible solution for the LP-relaxation of the RKP-R and must be optimal, since the same proof holds in this case.

If we want to calculate the LP-relaxation by using a simple algorithm, we can use the pseudo-code in Algorithm 7.

Algorithm 7 LP Relaxation Value for RKP-R

```
Value z ← 0
Sort items on ratio so that c_i/a_i > c_{i+1}/a_{i+1}
for Scenario s = |S| to 0 do
  Size a ← 0
  for Item i = 1 to |I| do
    if a + a_i ≤ b_s then
      z ← z + p_s c_i
      a ← a + a_i
    else
      if a < b_s then
        z ← z + (b_s − a)/a_i p_s c_i
        a ← b_s
      end if
    end if
  end for
end for
return z
```

5.4.4. Lagrangean Relaxation Upper-bound

When we take a look at the ILP of the RKP-R, we can move the capacity constraints into the objective by rewriting by introducing Lagrangean multipliers λ_s for each scenario s. Consider the ILP for the RKP-R.

$$
\begin{array}{llll}
\max & \sum_{i\in I} p_0 c_i x_i + \sum_{s\in S}\sum_{i\in I} p_s c_i y_i^s & & \\
\text{s.t.} & \sum_{i\in I} a_i x_i & \leq b_0 & \\
& \sum_{i\in I} a_i x_i & \leq b_s & \forall s \in S \\
& x_i - y_i^s & \geq 0 & \forall s \in S, \forall i \in I \\
& x_i, y_i^s & \in \{0,1\} & \forall s \in S, \forall i \in I
\end{array}
$$

The probability of our initial solution is denoted by p_0. Since we deal with probabilities, we may assume $p_0 + \sum_{s\in S'} p_s = 1$, i.e. all probabilities sum to 1. Now let us consider a different formulation for the RKP-R, using $\bar{y}_i^s$ variables that represent the removal of an item i in a scenario s, instead of retaining an item i in scenario s. Using the principle that the probabilities sum to 1, we can rewrite the object in such a fashion that we take the unweighted sum of the profits in the initial solution and subtract the weighted profits of the items that are removed in each scenario. Doing this, we get the following formulation:

$$\begin{array}{llll}
\max & \sum_{i\in I} c_i x_i - \sum_{s\in S}\sum_{i\in I} p_s c_i \bar{y}_i^s & & \\
\text{s.t.} & \sum_{i\in I} a_i x_i - \sum_{i\in I} a_i \bar{y}_i^s & \leq b_s & \forall s \in S \\
& \sum_{i\in I} a_i x_i & \leq b_0 & \\
& x_i - \bar{y}_i^s & \geq 0 & \forall s \in S, \forall i \in I \\
& x_i, \bar{y}_i^s & \in \{0,1\} & \forall s \in S, \forall i \in I
\end{array}$$

We can derive a Lagrangean Dual Problem by moving the capacity constraints into the objective using Lagragian multipliers $\lambda_0, \lambda_1, \ldots \lambda_{|S|}$. By doing this, we modify the objective to

$$\sum_{i\in I} c_i x_i - \sum_{s\in S}\sum_{i\in I} p_s c_i \bar{y}_i^s + \sum_{s\in S} \lambda_s (b_s - \sum_{i\in I} a_i x_i + \sum_{i\in I} a_i \bar{y}_i^s) + \lambda_0 (b_0 - \sum_{i\in I} a_i x_i)$$

To make things more readable, we introduce an extended set S' that contains all the scenario indices from S and the index 0 for the initial solution as well. So, $S' = S \cup \{0\}$ Using this set S', we can simply the objective to

$$\sum_{i\in I} c_i x_i - \sum_{s\in S}\sum_{i\in I} p_s c_i \bar{y}_i^s + \sum_{s\in S'} \lambda_s (b_s - \sum_{i\in I} a_i x_i + \sum_{i\in I} a_i \bar{y}_i^s)$$

We will simplify this formulation to

$$\sum_{i\in I} (c_i - \sum_{s\in S'} \lambda_s a_i) x_i + \sum_{s\in S}\sum_{i\in I} (-p_s c_i + \lambda_s a_i) \bar{y}_i^s + \sum_{s\in S'} \lambda_s b_s$$

The remaining constraints in the Lagrangean Dual Problem state that we can only remove an item for recovery for scenario s if it was taken initially. The goal is to choose positive values for the Lagrangean multipliers in such a fashion that the optimal solution for the fixed multipliers is minimized. Since the remaining constraints are very simple, it is possible to reduce the problem to a much simpler form.

If we apply the Lagrangean relaxation on the capacity constraint of the normal KP, the optimal value of dual multiplier of the capacity constraint is the ratio of the split item, which is discussed in [MST90]. The split item is the first item that doesn't fit completely if we take the items by best ratio. Now let us say that $\hat{i}_s$ is the split item for scenario s. We get the minimized value for the Lagrangean Relaxation if we put multiplier $\lambda_s = p_s \frac{c_{\hat{i}_s}}{a_{\hat{i}_s}}$, which is quite similar to the value for the multiplier in case of the normal KP. The proof for this statement can be found under Section C.3.

Of course, it is also possible to apply Lagrangean relaxation to the recovery constraints. This approach is discussed in Section 5.4.6.

5.4.5. Surrogate Relaxation Upper-bound

We will now consider the Surrogate Relaxation, introduced by [Glo68]. Suppose we have a couple of constraints on a problem, with indices j, where constraints have the form

$\sum_i a_i^j x_i \leq b_j$, with x_i as the ith variable and a_i^j as a factor in the linear constraint. Now we will combine a couple of constraints with indices in J' to a single constraint by using a surrogate factor μ_j for each of these constraints. The form of the resulting constraint is $\sum_{j \in J'}(\mu_j \sum_i a_i^j x_i) \leq \sum_{j \in J'} \mu_j b_j$, i.e. we just add all the constraints to each other after multiplying each constraint with a certain factor μ_j for that constraint. Just like with the Lagrangean Relaxation, the parameters μ_j should be chosen in such a fashion that the optimal value of the objective of the resulting problem is minimized (in case the original problem is a maximization problem and vice versa).

$$
\begin{array}{rclll}
LRKP - R & = & \max & \sum_{i \in I} p_0 c_i x_i + \sum_{s \in S} \sum_{i \in I} p_s c_i y_i^s & \\
& & \text{s.t.} & \sum_{i \in I} a_i x_i \leq b & \\
& & & \sum_{i \in I} a_i y_i^s \leq b_s & \forall s \in S \\
& & & x_i \geq y_i^s & \forall i \in I, \forall s \in S \\
& & & x_i, y_i^s \in \{0,1\} & \forall i \in I, \forall s \in S
\end{array}
$$

When we apply the surrogate relaxation to the $LRKP - R$, we get the following:

$$
\begin{array}{lll}
\max & \sum_{i \in I} p_0 c_i x_i \sum_{s \in S} \sum_{i \in I} p_s c_i y_i^s & \\
\text{s.t.} & \sum_{i \in I} \mu_0 a_i x_i + \sum_{s \in S} \sum_{i \in I} \mu_s a_i y_i^s \leq \mu_0 b + \sum_{s \in S} \mu_s b_s & \\
& x_i \geq y_i^s & \forall i \in I, \forall s \in S \\
& x_i, y_i^s \in \{0,1\} & \forall i \in I, \forall s \in S
\end{array}
$$

When we consider this surrogate relaxation of RKP-R problem, we are left with a problem that only has a single capacity constraint and a lot of constraints that state that an item can only be chosen after recovery if that item is selected initially. In this model we have multiple variables for each item: a variable that represents whether the item is chosen in the initial solution and a variable for each scenario that represents whether the item is chosen in the recovery solutions. Since the recovery variables of a single item are only dependent on the selection of the initial copy of that item, we have a single precedence tree on the variables of each item with the initial variable as the root and the recovery variables as the children. We will reduce this problem to a Precedence Constrained Knapsack Problem by considering all these separate variables as separate items. We add a single dummy item with weight and profit 0, and we add precedence constraints from this dummy node to all the roots of the trees we had. Now we are left with a single precedence tree. Thus, by using the surrogate relaxation of the RKP-R, we get the Precedence Constrained Knapsack Problem where the precedence constraints form a tree. An example for a RKP-R with 2 scenarios beside the initial scenario and 3 items is presented in Figure 5. We discussed an efficient algorithm for this Precedence Constrained Knapsack Problem on trees in Section 5.2.3.

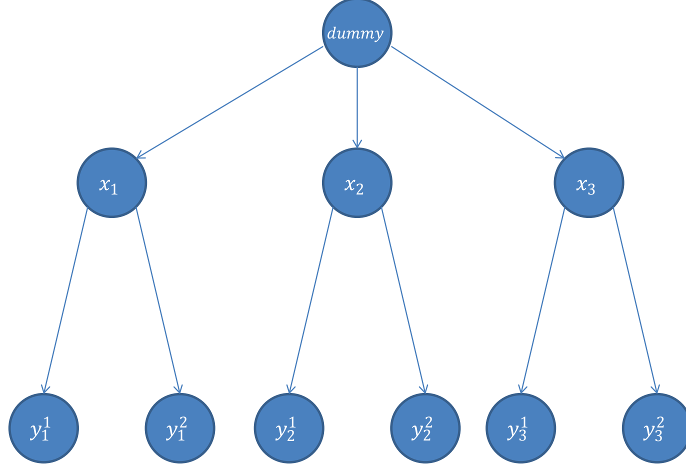


Figure 5: PCKP-tree for a RKP-R surrogate relaxation with 3 items and 2 scenarios

5.4.6. Recovery Relaxation Upper-bound

Another way to calculate an upper-bound for the RKP-R is by removing the recovery constraints (that have the form $y_i^s \leq x_i$). Suppose we remove all recovery constraints from the problem. We are left with a problem where we must select an item-set for each separate scenario, where we can disregard the relationships between the item-sets. We can solve this problem by calculating a table with the basic Dynamic Program from section 5.2.1. We can use a single table to determine the optimal knapsack for each of the scenarios. The objective value is the sum of the values of the optimal knapsack for each scenario multiplied by the probability of those scenarios.

Beside completely removing the constraints, another option is to apply Lagrangean relaxation to them. We can do this by using a Lagrangean dual λ_i^s for each item i for each scenario s and moving the recovery constraints $x_i \geq y_i^s = x_i - y_i^s \geq 0$ into the objective function. The new objective function becomes

$$\max \quad \sum_{i \in I} p_0 c_i x_i + \sum_{s \in S} \sum_{i \in I} p_s c_i y_i^s + \sum_{i \in I} \sum_{s \in S} \lambda_i^s (x_i - y_i^s - 0)$$

equals

$$\max \quad \sum_{i \in I} (p_0 c_i + \sum_{s \in S} \lambda_i^s) x_i + \sum_{s \in S} \sum_{i \in I} (p_s c_i - \lambda_i^s) y_i^s$$

Now let us consider this objective. We want to find the lowest possible upper-bound (which is the solution to the Lagrangean dual problem). We will start will all λ_i^s at 0. We will define $\uparrow \lambda_i^s$ as the increase of λ_i^s currently under consideration. We will show that it is only necessary to increase the lambda λ_i^s if our current solution has i selected for the initial item-set, but not for the recovery item-set of scenario s (i.e $x_i = 0$ while $y_i^s = 1$).

$x_i = 0, y_i^s = 0$ If we raise λ_i^s, it is clear the objective cannot decrease.

$x_i = 1, y_i^s = 0$ If we raise λ_i^s in this case, the objective value will increase, since y_i^s was 0 and its costs were decreased, while x_i was already 1 and its costs were raised.

$x_i = 1, y_i^s = 1$ If we raise λ_i^s in this case, the objective value will at least be equal to the value of the the current solution. Since λ_i^s is raised, x_i gains higher costs so it will remain in the current solution. Because the costs of y_i^s are lowered, there are two possibilities: y_i^s stays 1 or y^s becomes 0. If y_i^s stays 1, the value of solution doesn't change, because the exact cost decrease for y_i^s is added to the solution value as an cost increase for x_i. If y_i^s becomes 0, there are two options: the only y_i^s changes, or other elements of the item-set for scenario s change (so, there is at least some j such that y_j^s was 0 and becomes 1).

Only y_i^s changes Now suppose y_i^s becomes 0 and nothing else changes: the objective value is increased by $\uparrow \lambda_i^s$ since the costs of x_i increase by it, while it decreases by $\min\{\uparrow \lambda_i^s, p_s c_i - (\lambda_i^s + \uparrow \lambda_i^s)\}$, which must be smaller or equal to $\uparrow \lambda_i^s$. This implies the objective value is not decreased in this case.

Item-set for s changes The objective value will still increase by $\uparrow \lambda_i^s$. Now since a new solution was chosen for the scenario s item-set, this item-set must yield a better value than the previous item-set with the decreased costs for item i. If we had retained the solution with item i, the value of the solution would have been decreased by $\uparrow \lambda_i^s$. Since the new item-set dominates that solution, the value of the solution will be decreased by less than $\uparrow \lambda_i^s$, so the total objective value will increase.

$x_i = 0, y_i^s = 1$ If we raise λ_i^s, while the item-sets remain the same, the objective value is decreased, since x_i remains zero 0. If the item-sets do change, there are two possibilities: y_i^s becomes 0 or x_i becomes 1. If y_i^s becomes 0, there must be some item-set for scenario s that was dominated before by the item-set with item i, but now itself dominates that item-set. The difference in value between these item-set is the decrease of the objective value. If x_i becomes 1, the initial item-set is dominated by an item-set that now includes item i. Suppose we increased λ_i^s exactly enough such that the value of the item-set with i is equal to the previous initial item-set. This implies that the value of the initial item-set didn't improve, while the value of the recovery item-set for scenario s was decreased, so the objective value was decreased.

As a result, the only case where we want to increase a λ_i^s is the case where y_i^s is 1 while x_i is 0, since the other cases can not possibly decrease the objective value. This principle can be used to design a subgradient descent method [SKR85].

5.5. Linear Programming Techniques

5.5.1. Separate and Combined Phases Decompositions

Now suppose we use the decompositions presented in Sections 3.2 and 3.4.2 for the RKP-R. Let us first consider the separate recovery decomposition. In this decomposition,

a column in our restricted master problem represents either a single initial knapsack, or a recovery knapsack. Since we use the LP-relaxation of the restricted master problem to find new variables, let us consider the cases where the optimal solution to the LP relaxation leaves us with a fractional solution that is infeasible for the original problem.

Suppose we have 2 scenarios and 3 items with $a_1 = 3, a_2 = 3, a_3 = 10$ with $c_i = a_i$, with $b_0 = 10$ and $b_1 = 3$ and $p_0 = p_1 = 0.5$. Clearly, the optimal solution is the case where we choose item 3. However, when we may use a fractional solution with the separate recovery decomposition, we can choose a solution where we take items 1 and 2 for 0.5 and choose the solution where we take item 3 for 0.5. This gives us the opportunity to choose a solution with item 1 as the recovery solution for scenario 1 with value 0.5, but also choose a solution with item 2 as the recovery solution for scenario 1. While the value of the optimal integer solution is 5, this fractional solution gives a value of 5.5.

Another example is the case where we have items with sizes $a_1 = 15, a_2 = 19, a_3 = 6, a_4 = 7$ with $c_i = a_i$ and scenarios $b_0 = 36$ and $b_1 = 22$. If calculate the optimal solution for the LP-relaxation of the separate recovery decomposition, we get the item-set $\{1, 2\}$ with value $\frac{2}{3}$ for the main solution and the item-set $\{2, 3, 4\}$ with value $\frac{1}{3}$ for the main solution. This allows us to take item-sets $\{2\}$, $\{1, 3\}$ and $\{1, 4\}$ each with value $\frac{1}{3}$ for the scenario. However, the item-sets $\{1, 3\}$ and $\{1, 4\}$ are not even a proper subset of any of the sets we chose for the main scenario - they are a subset of the combination of all sets chosen for the main scenario.

Now suppose we will consider the combined recovery decomposition. With this decomposition, we will look for solutions that combine the initial and recovery solution for our columns. This way, it is impossible that we choose some recovery item-set that is not a subset of a selected initial item-set. However, it is still possible to get a fractional solution for our LP-relaxation of the problem.

Let us again consider a subset sum instance, with items $a_1 = a_2 = 4$ and items of size $a_3 = a_4 = 6$ with $a_i = c_i$. Our scenarios are $b_0 = 13, b_1 = 10, b_2 = 8$ with $p_0 = p_1 = p_2 = \frac{1}{3}$. The best solution for scenario 1 is the situation where we have $\{1, 3\}$, while the best set for scenario 2 is the situation where we have $\{1, 2\}$. Now we can take two columns with $\{1, 3\}$ and $\{2, 4\}$ as both an initial solution and scenario 1 with value 0.5. Since we have two different item-sets, we can now choose a column with $\{1, 2\}$ for scenario 2 with value 0.5 and a column with $\{3, 4\}$ as a initial solution and $\{3\}$ for scenario 2 with value 0.5. This gives us a situation where the initial item-sets chosen are not equal.

To conclude, these decomposition may not give an optimal solution without resorting to branch and price. A trivial idea is to branch on an item, but this still leaves a lot of possibilities. We can first branch by forbidding an item and after we explored that branch, make it mandatory, but also vice versa. There is also the question about the kind of item we will choose - do we choose an item depending on ratio, profit or costs. And this also leaves the question whether we take the greatest or the smallest item with regard to this measure. The question which branching strategies will work is explored through experimentation in Chapter 6.

5.5.2. Scenario Column Decomposition

Beside the Separate and Combined Phases Decompositions, we can also consider another type of decomposition. For each scenario, we may select each item exactly once. Instead of choosing an item-set for each scenario, we can also consider all possible sets of scenarios for each of the items and choose a single scenario set for each item. If we don't use an item at all, we will choose an empty set of scenarios for the item. If we choose it scenario 2 and 4, we will use scenario set $\{2,4\}$ for the item. Suppose x^i_σ is a binary variable that states whether we will use a subset $\sigma \subseteq S$ for item i. We define 2^S as the superset of S, which is the set that contains all possible subsets of S. We can now create an Integer Linear Program of the following form:

$$\begin{array}{lll}
\max & \displaystyle\sum_{\sigma\in 2^S}\sum_{i\in I}(p_0+\sum_{s\in\sigma}p_s)c_i x^i_\sigma & \\
\text{s.t.} & \displaystyle\sum_{\sigma\in 2^S} x^i_\sigma \leq 1 & \forall i\in I \\
& \displaystyle\sum_{\sigma\in 2^S}\sum_{i\in I}[s\in\sigma]a_i x^i_\sigma \leq b_s & \forall s\in S
\end{array}$$

If we solve this formulation as an ILP, we get a solution that is feasible in the original form: for each item we have a set of scenarios. We can set each x_i from the original problem to 1 if a column for that item was selected. We set each y^s_i of the original problem to 1 if that item i has a column selected with a non-zero factor in the constraint row for s. Since the capacity constraints are part of this approach, the derived solution will be feasible for the original problem. This correspondence can be reversed to see that a solution for the original problem can also be translated to a solution for this formulation, while remaining feasible. However, the LP-relaxation of this formulation gives the same value as the LP-relaxation of the original formulation of the RKP-R.

Theorem 5.4. *The optimal value of the LP-relaxation of the Scenario Column Decomposition is equal to the LP-relaxation of the original ILP formulation of the RKP-R.*

Proof. Let us consider the ILP-formulation of the RKP-R and its LP-relaxation.

$$\begin{array}{llll}
\text{LRKP-R} = & \max & \displaystyle\sum_{i\in I}p_0c_ix_i+\sum_{s\in S}\sum_{i\in I}p_sc_iy^s_i & \\
& \text{s.t.} & \displaystyle\sum_{i\in I}a_ix_i \leq b & \\
& & \displaystyle\sum_{i\in I}a_iy^s_i \leq b_s & \forall s\in S \\
& & x_i \geq y^s_i & \forall i\in I,\forall s\in S \\
& & x_i, y^s_i \in\{0,1\} & \forall i\in I,\forall s\in S
\end{array}$$

In Section 5.4.3 we discussed how to calculate the optimal solution of the LP-relaxation of this LRKP-R. We use Algorithm 7 to do this. This algorithm starts with sorting the items by decreasing ratios and greedily adds items in this order to the knapsack of each scenario. The item that does not fit entirely, is added fractionally so that the knapsack

is filled to its exact capacity. Since the order of the items is the same for each scenario, the recovery constraint holds automatically.

If we consider the LP-relaxation of the original problem, each scenario item-set will contain all items sorted by ratio up until the split item for that scenario, while the split item is only partly in the knapsack. We start with the item i with the highest ratio and find a variable x^i_σ for that item with all scenarios σ, and fix it to 1. We continue doing this for the different items, until we reach the split item for the smallest scenario s. We will now consider two variables x^i_σ and $x'^i_{\sigma-s}$. Now suppose that we have b'_s capacity left in our capacity constraint for scenario s. We will set the value of x^i_σ to $\frac{b'_s}{a_i}$ and the value of $x'^i_{\sigma-\{s\}}$ to $1-\frac{b'_s}{a_i}$. We continue using all scenarios except the smallest, until we reach the next split item for the scenario which is now the smallest. Again, we repeat the procedure.

It is not very difficult to see that this yields a solution that is exactly the same as the solution to the LP-relaxation of the LRKP-R.

□

Therefore, we may conclude that this approach is not practical to use in practice, since the ILP is difficult to solve and we have a more efficient way to calculate the LP-relaxation.

5.5.3. Cutting Plane Techniques

Suppose we have an ILP formulation of the original KP (i.e. $\{\max \sum_{i\in I} c_i x_i | \sum_{i\in I} a_i x_i \leq b, x_i \in \{0,1\} \forall i \in I\}$). When we use the LP-relaxation of this problem, it is possible we find a solution where all items have an integer value. In such a case, the LP-relaxation is also feasible for the original KP. However, in many cases, the LP-relaxation will have a fractional item in its solution. Now suppose we have some item-set $I' \subseteq I$ as the item-set for the LP-relaxation, where an item i is in I' if it is has a non-zero value in the solution to the LP-relaxation. Since there is a fractional item in I', we know that the solution to the original KP can't have all items in I'. Suppose the solution to the KP can contain more items from I' than $|I'|-1$. This must imply that all items from I' are in the solution, but the sum of the sizes of these items exceed the capacity constraint, so that solution must be infeasible. We can generate a new constraint: $\sum_{i\in I'} x_i \leq |I'|-1$. This constraint states that the number of items from I' in the solution to the problem, can be at most the number of items in I' minus one.

The additional constraints that are added to the LP-model to cut off solutions that are not feasible in the ILP-model, are called Cutting Planes. The technique of Cutting Planes was introduced in the 1960's by [Gom58]. within a Branch-and-Cut approach, a technique that combines Branch-and-Bound with Cutting Planes.

Cutting Planes are an important technique to reduce the intergrality gap and can also be applied to recoverable variants of the KP. Let us consider the RKP-R again. Suppose we have a solution to the LP-relaxation, where some scenario has a fractional item-set. Let us call I' the set of indices of items that are at selected for at least a part. We can

now generate a cut that states $\sum_{j \in I'} x_i \leq |I'| - 1$, or $\sum_{j \in I'} y_i^s \leq |I'| - 1$, which forbids the current fractional item-sets, but allows all other item-sets.

However, we choose to focus on column generation in our experimentation. While this is a possible way to solve this specific problem, we have no way of telling it is practical to do so.

6. Experiments

6.1. Introduction

In this Section, we will conduct computational experiments using the algorithms for the RKP-R presented in the previous sections. To be able to apply these algorithms, we need problems to apply them to. We discuss the generation of problem instances in Section 6.2. In Section 6.3, we discuss the experiments themselves.

Our experiments will consist of three phases. In the first phase we will test a couple of algorithms on the different types of instances to find the types that are the most difficult. In the second phase we will test a lot of different algorithms with varying parameters against a small set of instances of the types selected after the first phase to determine which ones work good and fast, and which ones don't. In the third phase we will test the best algorithms from the second phase against a larger data-set, to determine the effect adding more items and more scenarios to certain instances on the running time and in case of local search algorithms on the optimality gap.

6.2. Problem instances

Since some of the algorithms have a running time of $O(b^2)$ in the maximum knapsack bound b and our local search algorithms use a dynamic programming algorithm in which $b_{\max} - b_{\min}$ is an important factor, we want to keep our value of $b_{\max}$ small enough to be able to perform a lot of experiments. At a later stage we will increase these values to search for the limits of our algorithms.

However, these algorithms could be improved by including the balanced knapsack technique. In such a way, the running time is mostly influenced by the size of the largest item and less by the value of $b_{\max}$.

We will first discuss how to generate instances. To generate an instance, we need a set of items and a set of scenarios. We introduce a number of parameters for the generation of instances. For the generation of our item-sets, we introduce a parameter M that defines a method, a parameter R (which is usually an indication for size of the items) and a parameter n that defines the number of items. For the generation of the sets of scenarios, we have a parameter B, which is the sum of all items in the item-set we are currently considering, a parameter k that defines the range of scenario probabilities and a parameter ρ that defines the range of scenario capacities.

6.2.1. Generating Items

We begin by generating items for our instance. A lot of research has been done to find difficult instances of the regular knapsack problem. For our experiments we will use the research done by [Pis05] (also available on page 150 of [KPP04]) to create our problem instances. The rest of this section is a summary of this work, but it is included for reasons of convenience. There are a number of important instance classes resulting from this research, each dependent on a random range parameter R:

- *Uncorrelated instances* Both a_i and c_i are chosen randomly from $[1, R]$. These are often easy to solve

- *Weakly correlated instances* a_i is chosen randomly from $[1, R]$ and c_i is chosen randomly from $[a_i - R/10, a_i + R/10]$. These instances have a strong correlation between items, despite of their name. It can represent real life data in such a way that in general the return of an investment is proportional to the sum invested, proportional to some small variations.

- *Strongly correlated instances* a_i is chosen randomly from $[1, R]$ and $c_i = a_i + R/10$. It represents real life data in the sense that the return of an investment is proportional to the sum invested, but there is a fixed charge for each investment. The problems are hard to solve because in general there is a large gap between the continuous and integer solution of the problem and it is hard to solve the problem in such a fashion that there is no slack in the capacity constraint.

- *Inverse strongly correlated instances* c_i is chosen randomly from $[1, R]$ and $a_i = c_i + R/10$. These are like the strongly correlated instances, but the fixed charge is negative.

- *Almost strongly correlated instances* a_i is chosen randomly from $[1, R]$ and c_i is chosen randomly from $[a_i + R/10 - R/500, a_i + R/10 + R/500]$. These problems have both some kind of a fixed charge and also some noise. They represent both the properties of strongly and weakly correlated instances.

- *Subset Sum instances* a_i is chosen randomly from $[1, R]$ and $c_i = a_i$. These instances are often difficult to solve, because the simple upper-bound on the value of and instance is always equal to the capacity constraint.

Besides these general types of knapsack instances, [Pis05] also presents a couple of difficult instances with small coefficients. We give a short description.

- *Spanner instances* $span(v, m)$ A spanner-set takes the size of the spannerset v, the multiplier limit m and some item distribution as inputs. A set I of v items is generated according to our input distribution. The set is normalized to I' by dividing sizes and profits by $m + 1$, i.e. $a_{i'} = a_i/m + 1$ and $c_{i'} = c_i/m + 1$. The set is generated by repeatedly choosing an item k from I' and a multiplier μ from the interval $[1, m]$. The resulting item has $a_i = \mu * a_{k'}$ and $c_i = \mu * c_{k'}$. Computational experiments have shown that smaller value of v, for example $v = 2$ gives hard instances.

- *Multiple strongly correlated instances* $mstr(k_1, k_2, d)$. Each item i gets a size a_i chosen randomly from $[1, R]$. If a_i is divisible by d, $c_i = a_i + k_1$, otherwise $c_i = a_i + k_2$. The weights a_i from the first group will be multiples of d, so using only these weights you can only fill up to $d\lfloor c/d \rfloor$ of the capacity. To obtain a completely filled knapsack, items from the other group must be chosen. Computational experiments have shown

that very difficult instances can be obtained by $mstr(3R/10, 2R/10, d)$. $d = 6$ gave the most difficult instances, but values between 3 and 10 work good.

- *Profit Ceiling instances* $pceil(d)$ Sizes a_i are randomly chosen from $[1, R]$ and profits are $c_i = d\lceil a_i/d \rceil$. In experimental results, $d = 3$ was a good parameter to create difficult instances.

- *Circle instances* $circle(d)$ Sizes a_i are randomly chosen from $[1, R]$ and profit c_i becomes $d\sqrt{4R^2 - (w - 2R)^2}$. The profits can be seen as a function of the weights that forms an ellipsis.

6.2.2. Generating Scenario's

In the research of [Pis05], 100 regular knapsack instances were generated for a single set of items. The capacity of each instance with instance number $h = 1, ..., 100$ was chosen as $b_0 = \frac{h}{101} \sum_{i \in I} a_i$. Since our scenarios have multiple capacities with probabilities, we will use a different way to generate the scenarios. Two properties are important: the spread of these probabilities and the spread of the capacities. We will express the spread of the probabilities with a single integer number k, such that the initial probabilities p'_s of our scenarios $s \in S$ are initially set to a random value from the range $[1, k]$. After all our scenarios we will scale our scenarios, such that $p_s = \frac{p'_s}{\sum_{s \in S} p'_s}$. If we choose a value $k = 1$, the probability distribution of the scenarios will be uniform (i.e. each scenario has a probability $\frac{1}{|S|}$).

For the values of the capacities we know that any instance with a $b_{max} < a_{min}$ is trivial, as well a each instance with $b_{max} \geq \sum_{i \in I} a_i$. This means that we should look for our capacities in the range $[a_{min}, \sum_{i \in I} a_i - 1]$. To make our range more readable, let us define an upper-bound on the capacities $B = \sum_{i \in I} a_i$. Since have a capacity that is very close to $\sum_{i \in I} a_i - 1$ means that almost all items will be taken and a capacity of a_{min} means that almost all items won't be taken, it is expected that more interesting instances will have capacities somewhere in the range of $[\frac{1}{4}B, \frac{3}{4}B]$. For a given set of items and a given ratio $\rho \in [0, ..., 1]$, we will create three instances. One with capacities random chosen from the range $[\rho B, B]$, on with capacities in the range $[\rho \frac{3}{4} B, \frac{3}{4} B]$ and one with capacities in the range $[\rho \frac{1}{2} B, \frac{1}{2} B]$.

6.2.3. Generating Instances

In short, we generate instances using the following steps:

1. Choose a number of items n, a number of scenarios s, choose a method to generate items M with parameter R and possibly additional parameters, choose a spread for the scenario probabilities k and choose a spread ratio for the scenario capacities ρ.

2. Use M and its parameters to generate an item-set I with n items.

3. Calculate B such that $B = \sum_{i \in I} a_i$.

4. *(Optional)* If B exceeds a certain maximum value B_{max}, set B to B_{max}.
5. Generate a scenario set S_1 with normalized probabilities from $[1, k]$ and capacities in $[\rho B, B]$.
6. Generate a scenario set S_2 with normalized probabilities from $[1, k]$ and capacities in $[\rho\frac{3}{4}B, \frac{3}{4}B]$.
7. Generate a scenario set S_3 with normalized probabilities from $[1, k]$ and capacities in $[\rho\frac{1}{2}B, \frac{1}{2}B]$.
8. Add instances $(I, S_1), (I, S_2)$ and (I, S_3) to the pool of generated instances.
9. If we need more instances, restart.

6.3. Experiments

6.3.1. Choice of Parameters

Since we want to find the techniques that generate the hardest instances in general, we will choose a fixed value for R, the number of items and the number of scenarios. Since we want to create instances that can be solved rather fast, but are representative, we fix R to 30. We choose $k = 3$ and $\rho = 0.5$, so our scenarios will have enough diversity, but will probably be relevant in most cases.

6.3.2. Hardware and Software

All experiments were run on an Intel® Core™2 Duo 6400 @ 2.13 GHz, with 1GB of RAM, running Windows® XP Professional 64 bit, version 5.2, Build 3790, Service Pack 2. The algorithms where implemented in the Java Programming Language. The Java runtime used was the Java™SE Runtime Environment, build 1.6.0_17-b04, 32 bit edition. The SDK and compiler version used was 1.6.0_12. Linear Programs where solved using the ILOG® CPLEX 11.0 optimizer, 32 bit.

6.3.3. First Phase Experiments

For the first phase, we will generate a lot of different instances. Since we want to determine if certain classes are indeed more difficult for the algorithms we want to test, we need to choose some algorithms. Some preliminary testing has shown that the separate recovery Branch and Price algorithm works well in many cases where we branch on the item with the smallest value that is fractional in the current solution, including that item in the first case and excluding it in the second. Since we will also want to try local search algorithms, we will use the Branch and Price next to a Hill Climbing algorithm that does 10 random restarts.

We will use the following techniques to generate our instances of 20 items and 5 scenarios: *Uncorrelated, Weakly correlated, Strongly correlated, Inverse strongly correlated, Almost strongly correlated, Subset sum, span*$(2, 10)$ *from a weakly correlated set, span*$(2, 10)$ *from*

a strongly correlated set, $span(2,10)$ *from an uncorrelated set, multiple strongly correlated* $mstr(9,6,6)$*, profit ceiling* $pceil(3)$ *and* $circle(\frac{2}{3})$ instances, giving 12 types in total.

We will generate 500 item-sets for each instance class. Since our proposed method to generate scenarios will generate 3 instances for each item-set, this means our total number of instances will be 18000.

Instance	avg. ms	max ms	max nodes	max depth
Uncorrelated	17	781	129	8
Subset Sum	36	2,125	33	17
Uncorrelated $span(2,10)$	21	937	19	10
Weakly Correlated	45	2,297	1,025	11
Weakly Correlated $span(2,10)$	18	922	19	10
Strongly Correlated	147	3,188	31	16
Almost Strongly Correlated	334	204,625	31	15
Inverse Strongly Correlated	357	341,406	301	13
Strongly Correlated $span(2,10)$	19	1,547	29	15
$pceil(3)$	38	1,141	65	10
$mstr(9,6,6)$	96	4,984	67	10
$circle(\frac{2}{3})$	100	5,219	1,025	11

Table 3: First Phase - Branch and Price results

Instance	avg. ms	max ms	avg. $\frac{c}{c*}$	min $\frac{c}{c*}$
Uncorrelated	95	1,344	1	0.97
Subset Sum	55	234	1	1
Uncorrelated $span(2,10)$	35	297	1	0.88
Weakly Correlated	112	750	1	0.97
Weakly Correlated $span(2,10)$	34	266	1	0.98
Strongly Correlated	63	454	1	0.97
Almost Strongly Correlated	63	313	1	0.97
Inverse Strongly Correlated	119	657	1	0.94
Strongly Correlated $span(2,10)$	33	360	1	0.98
$pceil(3)$	78	547	1	0.99
$mstr(9,6,6)$	209	1,438	0.99	0.93
$circle(\frac{2}{3})$	238	1,859	0.99	0.9

Table 4: First Phase - HillClimbing results

The experiment took 2 hours and 45 minutes. Note that for each instance, the Java Virtual Machine was restarted, to make sure each program could start with a clean state of the memory. If we analyze the Branch and Price results in Table 3, we can see that the variations of the strongly correlated instances take the most time, both in terms of the maximum and average runtime. The inverse strongly correlated instances win

in terms of the greatest average runtime and the largest maximum runtime, while the almost strongly correlated instances take a lot of time as well. If we take a look at the Hillclimbing results in Table 4, we can see that the circle instances have the largest maximum runtime and the greatest average runtime. It is also interesting to see that the average deviation is almost 1 for all instances, but when we consider the minimum deviation, the uncorrelated $span(3, 10)$ instances perform the worst.

6.3.4. Second Phase Experiments

Since the almost strongly correlated instances and the inverse strongly correlated instances took the most time to solve with the Branch and Price algorithm, we will consider these instances for the second phase. The circle instances and the uncorrelated $span(3, 10)$ instances performed the worst with the Hillclimbing algorithm, with respect to running time and worst $\frac{c}{c*}$. Because of its classic status and its relation to the decision variant of the knapsack problem, we will also consider the subset sum instances in the second phase experiments.

This means we will consider only 5 instance classes in the second phase: the almost strongly correlated instances, the inverse strongly correlated instances, the circle instances, the spanned uncorrelated instances and the subset sum instances.

The goal of the second phase is to find out what algorithms are fast enough to do a large scale experiment with larger instances. We would also like to get some information about the global performance of the algorithms developed in the previous sections.

We have 12 different methods for branching in our Branch-and-Price algorithms. Any combination of smallest/greatest with ratio/value/size gives us 6 ways to choose an item from the set of fractional items. Since we may first include the item as well as exclude the item in our branches, we get 12 branching strategies in total. Since we have two Branch-and-Price algorithms, this leaves us with 24 Branch-and-Price strategies. To keep things fast enough, we use Hillclimbing (with 10 restarts) to find a starting solution for the Branch-and-Price.

The *Branch-and-Bound* algorithm also has the possibility to use the same branching strategies, which also gives us 12 strategies. Branch-and-Bound branches on the inclusion or exclusion of items in the initial item-set and uses Dynamic Programming (based on the basic approach presented in Section 5.2.1) to calculate the best recovery strategy in each node. We also use the LP-relaxation from Section 5.4.3 and a relaxation on the recovery constraint from Section 5.4.6 as upper-bounds.

The *Exact Dynamic Programming* algorithm from Section 5.3.5has no configuration options, but will also be used, since it is the only non-branching alternative to the other exact algorithms.

Iterative Dynamic Programming, the first approximation algorithm, is based on the technique discussed in Section 5.4.1 and uses a scheme comparable to the regular DP for knapsack, but calculates the optimal recovery in each state. Section 5.4.1 contains a counter-example that shows that this method is not guaranteed to give an optimal solution. Therefore, it may find different solutions depending on the ordering of the items and we may find better solutions if we consider each item more than once. We

may select different strategies to order and reorder the items after each iteration (during an iteration, each item is considered once). We will use strategies that sort the items according to size, profit and ratio (where the order is reversed after each iteration) and a configuration that shuffles the items randomly after each iteration. This gives us 4 different configurations.

Hillclimbing can be configured by choosing the number of random restarts and whether swap-operations should be considered or not. We will use 10 and 100 random restarts in combination with and without swaps. This also gives us 4 different configurations.

Simulated Annealing can be configured by selecting a cooling scheme, which can be either linear or exponential and its cooling parameter, the number of steps that should be done before cooling down, the starting energy, the ending energy, the maximum number of iterations without improvement and the probability to use hillclimbing to go to the nearest local optimum. We will fix the initial energy to 1, the minimum energy to 0.01 and the number of steps to take before cooling to 10. We will try the linear cooling scheme with parameters 0.1, 0.01 and 0.001, all with the local optimum step disable, except for the 0.01 parameter, which will also be tried with the local optimum step disabled, as well as giving it a probability of 0.01. We will use the exponential cooling scheme with the parameters 0.9, 0.99 and 0.999, where the local optimum step is disabled, except for the parameter 0.99 which will also be tried with the probability of 0.01 for the local optimum step. This gives us 8 configurations.

Tabu Search can be configured by selecting the length of the tabu-list, the number of steps that can be taken since the last improvement before ending the algorithm and whether or not swaps-operations should be considered. We will fix the number of steps without improvement to two times the number of items in the problem. With swapping enabled, all items can be swapped for each other in this number of steps. For the length of the tabu-list, we will try 3, 10 and 30. We will also try them with both swaps enabled and disabled, giving us a total of 6 configurations.

In summary, we will try the following algorithms:

- Branch and Price on the separate recovery decomposition* (12 configurations)
- Branch and Price on the combined recovery decomposition* (12 configurations)
- Branch and Bound* (12 configurations)
- Exact Dynamic Programming*
- Iterative Dynamic Programming (4 configurations)
- Hillclimbing (4 configurations)
- Simulated Annealing (8 configurations)
- Tabu Search (6 configurations)

The algorithms marked with a * are the exact algorithms (27 in total) , which give the value of the optimal solution, while the other only give an approximation (22 in total). The total number of configurations is 59.

Since the number of algorithm configurations is very large, we need an instance collection that has growing parameters to get an idea about the relation between input size and running time (both the number of items and the number of scenarios will need to grow). Also, we can expect certain algorithms to be very slow so we want very small instances as well as instances of a moderate size which large enough to get an idea about the behavior of the running times and deviations compared to the instance size. To accomplish this we will limit the number of items and the number of scenarios in our instances to rather small sets. The set of possible numbers of items for our instances is $\{5, 10, 15, 25\}$. The number of scenarios will be selected from $\{2, 4, 6, 8\}$. However, we exclude the combinations where the number of scenarios exceeds the number of items, i.e. combinations with 5 items and 6 or 8 scenarios. For each combination we will generate 20 item-sets and for each item-set we will generate three sets of scenarios, so 60 instances for each combination. Since we have 14 combinations and 5 instance types, this gives us 4300 instances for each algorithm, so we will have 253700 runs in total. We will limit the running time for each experiment to 3000 milliseconds, so the amount of time in which we can do our experiments will be acceptable.

The total experimentation time was 40 hours and 36 minutes. All results were processed into Tables 5 and 6. The first column tells us how many runs failed. A failed run means that the algorithm ran longer than 3000 milliseconds. Such experiments are ignored in the rest of the table. The next columns give us the average runtime in ms and the maximum runtime in ms. The third column give us the average deviation from the optimal value $\frac{c}{c^*}$ and the minimum deviation from the optimal value $\frac{c}{c^*}$ over all the solutions found by the algorithm. Note that the deviation is not relevant for the exact algorithms, since it should always be 1. Also note that we can only calculate the deviation if we have an exact solution. If no exact algorithm finds a solution, we cannot calculate the deviation for that particular instance.

The last two columns give a crude estimate on the average growth of the maximum runtime when additional items or scenarios are added to the instances. We begin with our item-set sizes $N = \{5, 10, 15, 25\}$ and our scenario sizes $K = \{2, 4, 6, 8\}$. Let us define a function $t(n, k)$ which gives the average running time in ms for n items and k scenarios. Now let us define a function $f(n) = \sum_{k \in K} \frac{t(n,k)}{|K|}$ that gives the average number of milliseconds t to find a solution for an item-set of size n. Likewise, we define $g(k) = \sum_{n \in N} \frac{t(n,k)}{|N|}$ for the number of scenarios. We calculate the average of the derivatives of these functions to get an idea about the behaviors of the running time of these algorithms.

When we consider the average running times, we can see that the Separate Phase Branch and Price, the Iterative Dynamic Programming and the Hillclimbing algorithms perform well, while their number of failures is also acceptable. The Combined Phases Branch-and-Price algorithms have a very large number of failures, because the large instances take too much time to solve. This holds even more for the Exact Dynamic Programming, which could be expected due to its exponential growth in terms of the number of scenarios. While the Simulated Annealing and Tabusearch algorithms have a good performance on average, considering their deviation, they can't beat the Hillclimbing

Technique	Failed (of 4300)	avg. ms	max ms	avg. nodes	max nodes	avg. f'	avg. g'
Combined Phases Branch-and-Price							
(0,1)-branching greatest ratio	1,400	425	2,953	1.13	17	47	275
(0,1)-branching greatest size	1,408	410	2,969	1.11	13	46	292
(0,1)-branching greatest value	1,416	402	2,969	1.12	17	46	290
(0,1)-branching smallest ratio	1,404	404	2,937	1.12	17	46	286
(0,1)-branching smallest size	1,413	407	2,969	1.11	13	44	280
(0,1)-branching smallest value	1,407	406	2,969	1.12	13	45	285
(1,0)-branching greatest ratio	1,407	417	2,969	1.12	17	45	286
(1,0)-branching greatest size	1,401	427	2,985	1.1	13	46	300
(1,0)-branching greatest value	1,406	421	2,984	1.11	17	46	294
(1,0)-branching smallest ratio	1,417	412	2,969	1.12	13	45	290
(1,0)-branching smallest size	1,395	434	2,938	1.13	13	47	305
(1,0)-branching smallest value	1,405	428	2,953	1.11	13	48	308
Separate Phase Branch-and-Price							
(0,1)-branching greatest ratio	190	101	2,938	2.97	261	4	29
(0,1)-branching greatest size	167	100	2,860	3.16	435	4	26
(0,1)-branching greatest value	167	95	2,844	3.16	521	4	28
(0,1)-branching smallest ratio	236	99	2,906	4.6	537	4	32
(0,1)-branching smallest size	225	108	2,937	4.35	1,019	5	32
(0,1)-branching smallest value	242	103	2,860	4.33	807	5	34
(1,0)-branching greatest ratio	128	107	2,563	3.27	122	5	31
(1,0)-branching greatest size	137	97	2,937	2.85	39	5	32
(1,0)-branching greatest value	122	101	2,906	3.33	831	5	30
(1,0)-branching smallest ratio	146	88	2,750	2.67	255	5	29
(1,0)-branching smallest size	135	90	2,828	2.93	257	4	26
(1,0)-branching smallest value	121	94	2,844	2.85	67	4	27
Branch and Bound							
(0,1)-branching greatest ratio	653	191	2,984	3,327	71,491	29	18
(0,1)-branching greatest size	792	215	2,922	5,104	160,676	33	46
(0,1)-branching greatest value	781	201	2,938	4,399	137,477	30	36
(0,1)-branching smallest ratio	713	207	2,984	3,791	156,121	36	4
(0,1)-branching smallest size	566	192	2,984	2,374	54,803	30	2
(0,1)-branching smallest value	595	211	2,985	2,949	129,550	35	-3
(1,0)-branching greatest ratio	748	148	2,984	2,994	37,669	18	1
(1,0)-branching greatest size	520	156	2,921	3,019	43,595	17	4
(1,0)-branching greatest value	547	166	2,906	3,308	78,057	20	-12
(1,0)-branching smallest ratio	190	111	2,906	1,281	33,321	11	7
(1,0)-branching smallest size	359	121	2,984	1,586	80,778	15	-0.22
(1,0)-branching smallest value	351	102	2,984	1,164	71,581	12	6
Exact Dynamic Programming	2,840	347	2,984	-	-	87	159

Table 5: Second Phase Experiments - Exact Algorithms

Technique	Failed (of 4300)	avg. ms	max ms	avg. $\frac{c}{c^*}$	min $\frac{c}{c^*}$	avg. f'	avg. g'
Hillclimbing							
10 random restarts	0	2	32	0.98	0.66	0.11	0.33
10 random restarts with swapping	0	55	1,391	0.99	0.81	5	10
100 random restarts	0	17	422	0.99	0.85	1	3
100 random restarts with swapping	222	337	2,985	1	0.88	33	70
Iterative Dynamic Programming							
Shuffle order	0	96	2,172	1	0.98	7	16
Order by ratio	0	82	2,531	1	0.99	8	20
Order by size	0	88	2,531	1	0.99	7	17
Order by value	0	86	2,266	1	0.99	7	16
Simulated Annealing							
Exponential cooling, factor 0.999	1,169	968	2,984	0.94	0	78	163
Exponential cooling, factor 0.99	35	302	2,969	0.88	0	24	59
Exponential cooling, factor 0.99 and Hillclimbing	81	319	2,968	0.98	0	25	63
Exponential cooling, factor 0.9	0	28	359	0.75	0	2	5
Linear cooling, cooldown 0.001	178	495	2,985	0.92	0	41	91
Linear cooling, cooldown 0.01	0	63	1,031	0.84	0	5	12
Linear cooling, cooldown 0.01 and Hillclimbing	0	109	2,657	0.98	0	8	22
Linear cooling, cooldown 0.1	0	3	47	0.61	0	0.14	0.45
Tabusearch							
Tabulist length 3	0	8	407	0.99	0.64	0.54	1
Tabulist length 10	0	6	485	0.99	0.64	0.42	1
Tabulist length 30	0	6	375	0.99	0.64	0.42	0.82
Tabulist length 3 with swapping	43	203	2,985	0.99	0.81	15	60
Tabulist length 10 with swapping	51	217	2,968	0.99	0.81	17	62
Tabulist length 30 with swapping	67	233	2,985	1	0.85	19	66

Table 6: Second Phase Experiments - Approximation Algorithms

and Iterative Dynamic Programming algorithms in terms of worst case performance, while being comparable in terms of the running time. The Branch and Bound approach gives surprisingly fast results and, while having more failures than the Separate Phase Branch-and-Price algorithm, is the second best exact algorithm. However, since adding items adds additional depth to the tree, we may expect an increasing running time when we additional items to our instances.

It is safe to conclude that Separate Phase Branch-and-Price, Hillclimbing and Iterative Dynamic Programming are the techniques to consider during the third phase. In case of the Iterative Dynamic Programming we will work with order-by-value. For Hillclimbing we will take the 100 random restarts, since it has a good value for the minimum deviation and never failed. For the Separate Phase Branch and Price technique we will work with the (1-0)-branching largest ratio principle, since it had the lowest maximum running time compared to other potential choices.

6.3.5. Third Phase Experiments

During the third phase, we will work with Separate Phase Branch-and-Price with (1-0)-branching on the item with the largest ratio, Hillclimbing with 100 random restarts and Iterative Dynamic Programming with order by value. The goal of the third phase is to get more precise information about the growth rate of the running time in relation to the number of scenarios and the number of items. For our instances, we will use the same types we used during the second phase, which gives us five instance classes. For our number of items, we will choose from $\{50, 75, 100\}$, while we will choose the number of scenarios from $\{2, 3, 4, 10, 20\}$. We use the same 5 different instance classes for our item-sets. For each combination of number of scenarios, items and each instance class, we generate 20 item-sets. This yields $3 \times 5 \times 5 \times 20 = 1500$ item-sets in total. We generate 3 sets of scenarios for each item-set, which gives us 4.500 instances in total. We will limit the running time of the algorithms to 4 minutes for each of these instances.

Analyzing the results from Tables 7, 8 and 9 we can make a few observations. When we take a look at the Branch and Price algorithm, we can see that the running time seems to increase reasonably when we raise the number of scenarios. While the number of failures is greater than the number of failures in case of the Hillclimbing algorithm, we have a guarantee for optimality. Additionally, the Hillclimbing algorithms needs a lot additional running time for each additional scenario or item added to the problem.

The average running time of the Iterative DP is dominated by both the Branch-and-Price and the Hillclimber. However, the number of failures of the Iterative DP dominates the number of failures of the Branch-and-Price, while the deviation from the optimal solution dominates the deviation of the Hillclimber. As expected, the average running time scales very predictable in the size of the input. It showed no deviation from the optimal solution, which suggests that the counterexample which showed the algorithm is not exact either does not occur often in random instances, or shows that these instances are difficult to the Branch and Price algorithm as well.

Now let us take a better look at the Branch-and-Price algorithm. In Table 10 we consider the Branch-and-Price results for the largest scenario sets. We see that there is a

Items	Scenarios	Failed (of 300)	avg. ms	max ms	avg. nodes	max nodes
50	2	2	686	56,312	1.56	68
50	3	12	2,724	53,454	1.7	25
50	4	46	3,799	58,688	2.6	35
50	10	125	3,295	53,483	2.29	35
50	20	144	1,473	38,766	1.4	17
75	2	45	1,702	42,984	1.12	7
75	3	111	2,365	43,485	1.59	43
75	4	123	1,648	33,515	1.54	41
75	10	169	712	29,454	1.18	17
75	20	177	155	3,203	1	1
100	2	114	1,695	47,531	1.05	5
100	3	173	703	24,781	1.16	11
100	4	176	964	46,172	2.03	59
100	10	213	468	34,547	1.39	25
100	20	210	103	2,703	1.13	13

Table 7: Third Phase - Seperate Phase Branch and Price (Totals)

Items	Scenarios	Failed (of 300)	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	1,204	0.98	0.84
50	4	0	181	1,203	0.98	0.83
50	10	0	268	1,407	1	0.94
50	20	0	309	1,515	1	0.84
75	2	0	339	10,906	0.98	0.75
75	3	0	538	12,484	0.99	0.78
75	4	0	596	9,687	0.99	0.7
75	10	0	1,022	11,953	1	0.77
75	20	0	1,450	12,984	1	0.74
100	2	0	887	19,656	0.98	0.66
100	3	0	1,257	25,578	1	0.86
100	4	0	1,783	32,625	1	0.8
100	10	0	3,546	34,703	1	0.8
100	20	0	4,546	37,312	1	0.94

Table 8: Third Phase - Hillclimbing 100 Restarts (Totals)

Items	Scenarios	Failed (of 300)	avg. ms	max ms	avg. $\frac{c}{c^*}$	min $\frac{c}{c^*}$
50	2	0	1,729	17,062	1	1
50	3	2	2,881	31,375	1	1
50	4	2	3,082	46,297	1	1
50	10	8	4,543	34,218	1	1
50	20	18	4,980	37,875	1	1
75	2	15	4,666	46,642	1	1
75	3	32	6,958	52,062	1	1
75	4	45	8,268	48,844	1	1
75	10	75	10,545	44,359	1	1
75	20	82	12,155	52,048	1	1
100	2	45	9,468	58,515	1	1
100	3	65	13,264	59,719	1	1
100	4	90	13,651	58,390	1	1
100	10	127	18,686	58,985	1	1
100	20	141	17,952	58,875	1	1

Table 9: Third Phase - Iterative DP (Totals)

Instance class	Items	Failed (of 60)	avg. ms	max ms	avg. nodes	max nodes
Subset Sum	50	3	36	78	1	1
	75	0	90	2,062	1	1
	100	19	78	156	1	1
Uncorrelated $span(2, 10)$	50	3	434	22,453	1.07	5
	75	15	91	1,860	1	1
	100	13	67	375	1	1
Almost Strongly Correlated	50	54	17,604	34,781	7.67	17
	75	56	145	204	1	1
	100	58	1,453	2,703	7	13
Inverse Strongly Correlated	50	44	4,864	38,766	1.25	5
	75	47	673	3,203	1	1
	100	60	-	-	-	-
$circle(\frac{2}{3})$	50	40	981	11,063	1.7	7
	75	59	203	203	1	1
	100	60	-	-	-	-

Table 10: Third Phase - Seperate Branch-and-Price (Details for 20 scenarios)

Algorithm	Items	Failed (of 60)	avg. ms	max ms	avg. nodes	max nodes
Branch and Price	50	3	36	78	1	1
	75	0	90	2,062	1	1
	100	19	78	156	1	1
Hillclimbing	50	0	185	391	1	1
	75	0	418	797	1	1
	100	0	739	1,469	1	1
Iterative DP	50	0	2,426	4,937	1	1
	75	0	8,098	13,781	1	1
	100	0	19,152	32,547	1	1

Table 11: Third Phase - Subset Sum Instances (Details for 20 scenarios)

large difference between the different instance classes: Subset Sum seems to be much easier than the other instance classes for the Branch-and-Price, since it often has the best running time compared to the Iterative DP and Hillclimbing, based on the information in Table 11.

To conclude, it is unclear to state which algorithm is the best. In case of subset sum instances, the Branch-and-Price algorithm seems a very good choice. If we have other instance types, the Branch-and-Price algorithm may give good results, but it may take too much time. If it takes too much time, we may consider both the Hillclimber or the Iterative DP, which seem to have a trade-off between running time and deviation from the optimal solution value.

7. Discussion

7.1. Decomposition Framework

In Sections 3.2 and 3.3 the ideas behind the Separate Recovery Decomposition and the Combined Recovery Decomposition for Recoverable Robust Problems were presented. The idea is to decompose these extended problems for multiple scenarios into multiple problems for a single scenario, since adapting algorithms for original problems to consider a single scenario is usually easier than adapting such an algorithm to consider all scenarios. The Separate Recovery Decomposition decomposes the problem into a separate problem for finding an initial solution and separate problems for finding a recovery solution for each scenario, but assumes it is possible to express the constraints on the recovery using linear constraints. The Combined Recovery Decomposition decomposes the problem into separate problems for finding both an initial and a recovery solution for a single scenario, considering the recovery constraints inside these problems instead of expressing them as linear constraints.

These decompositions are tried on some example problems. We examine the knapsack problem most thoroughly: in Section 4.1.1 we discuss the formal application of the decompositions to the problem. Additionally, we explore variants of the problem and techniques to solve these variants in Chapter 5. Also, computational experiments are conducted in Chapter 6.

The decomposition is also applied to the Weighted Independent Set problem. In Section 4.2 we present the result of applying both decompositions to an extended variant of the Weighted Independent Set problem. We also show how the combined problem derived from the combined recovery decomposition can be transformed into a single independent set problem, although at the cost of expanding the size of the problem instance by a constant factor. We also explore an extension of the shortest path problem and the much simpler classroom problem in Sections 4.3 and 4.1 and derive the combined recovery decomposition for these problems.

In all these cases, it is possible to derive pricing problems that can be reduced to the original problem in some form and for which we can apply original algorithms. This gives empirical evidence to support the idea that the framework is indeed capable to reduce recoverable robustness problems into multiple original problems that can be solved using specialized algorithms.

A potential drawback of the framework is the notion that possible similarities between the scenarios are ignored. It may be the case that this leads to some redundant work, when solving the pricing problems.

When we applied the Danztig-Wolfe decomposition in Section 3.4, we assumed all subproblems have a bounded region. If we want to consider unbounded problems as well, we will need to take extreme directions into account, which will yield a slightly different deriviation.

7.2. Knapsack Problems with Robustness

In Section 5.3.1 we introduced many recoverable robustness variants of the Knapsack Problem, using a three segment notation. Examples are the RKP-S, which limits the recovery to swapping, the RKP-RG which limits recovery to a greedy rule, the F-KP-RC which limits the recovery to removing a limited number of items and demands feasibility and RKP-R, which limits recovery to only allowing the removal of items.

Some of the discussed problems can be solved using a pseudo-polynomial Dynamic Program. This implies these variants are weakly NP-hard and are thus in the same complexity class as the regular Knapsack Problem. However, for the RKP-R, no pseudo-polynomial algorithm could be found and regular reductions from 3-partition fail (as shown in Section C.4), so the complexity class of this problem is still uncertain. These RKP-S with a bounded number of swaps seems also hard.

It is difficult to observe a pattern in the complexity of the different variants - where the RPK-RG can be solved using a technique that takes an equal amount of time as the regular KP, the RKP-S increases the amount of time required to solve the problem while remaining weakly NP-hard. Putting a bound on the maximum number of swaps seems to make the problem harder from an algorithm design perspective. Removing items without further restrictions also seems to be a hard problem, while putting a restriction on maximum number of items to be able to remove and simplifying the objective by demanding only feasible recovery solutions and optimizing the initial solution, makes the problem easier again, since a pseudo polynomial time algorithm exists. The structure of the complexity hierarchy of these robust knapsack variants seems to be counter-intuitive. It is an interesting, but difficult question what causes these varying difficulties for the different problems.

The smaller instances of RKP-R are solvable in practice using either the decomposition framework or an iterative improvement technique. Additionally, when we consider Subset Sum instances, the decomposition framework has the advantage. The question remains whether this behavior also occurs when we apply these decomposition techniques to other problems and if the process can be refined (for example by using better branching rules or other ideas).

7.3. Experiments

In Chapter 6, we presented three distinct stages of experimentation. During the first stage we consider different problem instances for the regular knapsack problem and try to find hard problems for the extended RKP-R. During the second stage, we considered many different algorithms to solve these hard instances and tried to find the best algorithms. During the third stage, we tried the best algorithms on larger instances to get some insight into the amount of time it takes to solve these problems.

At the end of the experiments, the random restart Hillclimbing and a branch-and-price algorithm based on the separate recovery decomposition show good results, while the dynamic programming techniques takes a lot of time when the number of items or scenarios is increased. This shows that the decomposition technique is not only interesting

from a theoretical point of view, but also has some practical significance.

7.4. Practical Implications

While the separate phase decomposition yields some practical results for the RKP-R, other robust knapsack problems could be solved using specific dynamic programs. In general, using such a specific algorithm is preferable in most cases, but in our case it was difficult to find an efficient and exact algorithm for the RKP-R. This gives some insight into the field of application of the framework: it can be used when the complexity of the robustness problem is too great to manage.

With regard to solving problems, the separate recovery decomposition has proven itself with regard to the robust knapsack problem. However, the combined recovery decomposition still has to prove itself, since it was very slow in case of the robust knapsack problems. However, the algorithm used to solve the pricing problem was quite slow, due to a running time quadratic in the capacity. In this regard, the demand robust shortest path problem seems a likely candidate for experimentation, because it derived positive weight shortest path problems for pricing, while the RKP-R derived knapsack problems for pricing with two scenarios that take significantly more time to solve than original knapsack problems with the same number of items and equal capacity. An advantage of the combined recovery decomposition is that it does not depend on the expressibility of the recovery constraints using linear equations.

7.5. Future Research

While the experiments on the knapsack show some promising results, a lot more can be done. More experimentation with these techniques on other problems seems like an interesting thing to do - especially the demand robust shortest path problem seems like a good candidate for further experimentation. We can also consider other options, like special instances of the weighted independent set problem - a variant of this problem in trees is also a candidate, since there is a polynomial time algorithm for these special instances. There is also some room to consider different variants of the knapsack problem - the RKP-S with a bounded number of swaps for recovery seems like a good candidate to explore. Experimentation using the results about the Lagrangian recovery relaxation from Section 5.4.6 and the surrogate relaxation from Section 5.4.5 is also an option regarding the Robust Knapsack Problems.

The question whether the RKP-R is weakly or strongly NP-hard remains. While the algorithm from Section 5.3.5 implies that the RKP-R is weakly NP-hard for a fixed number of scenarios, the question whether adding more scenarios to a problem instance makes it significantly more difficult is still open. We also spoke about the option of adding additional scenarios on the fly in case of minimax objectives, in Section 3.6. It might be interesting to develop some sort of sensitivity analysis method, that tells us what kind of scenarios are covered by the current solution (and thus can be added without disturbing our initial solution). Such considerations about the complexity or the sensitivity of some initial solution can be applied to both the knapsack variants explored in Chapter 5, but

also with respect to recoverable robustness problems in general. It is also interesting to look at ways to reduce the amount of redundant work the pricing problems perform on similar parts of the problem. Preprocessing methods that take this into account would be a valuable subject to research. The sections about core-methods for the Knapsack Problem in [KPP04] might be considered.

Also, the experimental results from Tables 5 and 6 show some promise for the usage of local search techniques that apply an exact recovery algorithm on the generated initial solutions. One can consider refining these experiments to give more guarantees on finding a good solution, or consider such techniques for different recoverable robustness problems.

The combined recovery decomposition deserves some attention as well. Its application on the demand robust shortest path problem gives nice theoretical results and seems like a good test-case for this approach. Additionally, it may be possible to move the costs of the initial solution into the pricing problems as well (by scaling them by a factor $\frac{1}{|S|}$ and moving the costs to the w^s_{pq} variables in the master problem transformation from Section 3.3), which may make applications to problems with non-linear cost functions possible as well.

8. Conclusion

When we consider our work from a theoretical point of view, we get nice decompositions when we apply them to a couple of classic problems. Robustness variants of the knapsack problem, the shortest path problem and the weighted independent set problem are all reduced to their original form in the pricing problems derived by the decomposition techniques. This allows us to apply well known algorithms to solve the recoverable robustness variations of these problems.

From a practical point of view, we have also shown that the separate recovery decomposition can be used to solve some problems of moderate size, containing 100 items and having 20 scenarios, in a matter of seconds. While there are also some instances of moderate size that cannot be solved in 4 minutes, the used techniques can possibly be refined, or used as an approximation method to find good instead of optimal solutions.

Our work paves the way for multiple paths of future research. The decomposition framework can be applied to other problems, producing interesting opportunities for further experimentation and analysis. Additionally, the multiple variants of the robust knapsack problem are interesting problems to study in itself, either because of the questions about the complexity of these problems, or simply because the problems provide challenges in an algorithmic sense.

References

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1 edition, February 1993.

[Bab79] L. Babai. Monte-carlo algorithms in graph isomorphism testing. *Université tde Montréal Technical Report, DMS*, pages 79–10, 1979.

[Bel57] Richard Bellman. *Dynamic Programming.* Dover Publications, 1957.

[BJN+98] 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):pp. 316–329, 1998.

[BJS04] Mokhtar S. Bazaraa, John J. Jarvis, and Hanif D. Sherali. *Linear Programming and Network Flows.* Wiley-Interscience, 2004.

[BKK11] Christina Büsing, Arie Koster, and Manuel Kutschka. Recoverable robust knapsacks: the discrete scenario case. *Optimization Letters*, pages 1–14, 2011. 10.1007/s11590-011-0307-1.

[BL97] John R. Birge and François Louveaux. *Introduction to Stochastic Programming.* Springer Series in Operations Research and Financial Engineering. Springer, July 1997.

[BS04] D. Bertsimas and M. Sim. The price of robustness. *Operations research*, pages 35–53, 2004.

[BTGN09] A. Ben-Tal, L. El Ghaoui, and A. Nemirovski. *Robust Optimization.* Princeton University Press, 2009.

[BTvDvL08] Hans Bodlaender, Richard Tan, Thomas van Dijk, and Jan van Leeuwen. Integer maximum flow in wireless sensor networks with energy constraint. In Joachim Gudmundsson, editor, *Algorithm Theory - SWAT 2008*, volume 5124 of *Lecture Notes in Computer Science*, pages 102–113. Springer Berlin / Heidelberg, 2008.

[CKPP98] Alberto Caprara, Hans Kellerer, Ulrich Pferschy, and David Pisinger. Approximation algorithms for knapsack problems with cardinality constraints. *European Journal of Operational Research*, 123:2000, 1998.

[Coo98] W. Cook. *Combinatorial optimization.* Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1998.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[DGRS05] 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.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390.

[dMVDH99] Olivier du Merle, Daniel Villeneuve, Jacques Desrosiers, and Pierre Hansen. Stabilized column generation. *Discrete Mathematics*, 194(1-3):229 – 237, 1999.

[DW60] George B. Dantzig and Philip Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):pp. 101–111, 1960.

[Fis81] Marshall L. Fisher. The lagrangian relaxation method for solving integer programming problems. *Management Science*, 27(1), 1981.

[GJ78] M. R. Garey and D. S. Johnson. " strong " np-completeness results: Motivation, examples, and implications. *J. ACM*, 25:499–508, July 1978.

[GL97] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[Glo68] Fred Glover. Surrogate constraints. *OPERATIONS RESEARCH*, 16(4):741–749, 1968.

[Gom58] Ralph E. Gomory. Outline of an algorithm for integer solutions to linear program. *Bulletin of the American Mathematical Society*, 64(5):275–278, September 1958.

[JN83] D. S. Johnson and K. A. Niemi. On knapsacks, partitions, and a new dynamic programming technique for trees. *MATHEMATICS OF OPERATIONS RESEARCH*, 8(1):1–14, 1983.

[Kar72] R. M. Karp. Reducibility Among Combinatorial Problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KPP04] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[LD60] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):pp. 497–520, 1960.

[LLMS07] Christian Liebchen, Marco Lübbecke, Rolf H. Möhring, and Sebastian Stiller. Recoverable robustness. Technical report, ARRIVAL-Project, August 2007.

[MST90] Martello, Silvano, and Paolo Toth. *Knapsack problems: algorithms and computer implementations.* John Wiley & Sons, Inc., New York, NY, USA, 1990.

[Pis99] David Pisinger. Linear time algorithms for knapsack problems with bounded weights. *Journal of Algorithms*, 33(1):1 – 14, 1999.

[Pis05] David Pisinger. Where are the hard knapsack problems? *Comput. Oper. Res.*, 32:2271–2284, September 2005.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach.* Prentice Hall, 2nd international edition edition, 2003.

[Sch02] A. Schrijver. On the history of the transportation and maximum flow problems. *Mathematical Programming*, 91(3):437–445, 2002.

[SKR85] N. Z. Shor, Krzysztof C. Kiwiel, and Andrzej Ruszcayǹski. *Minimization methods for non-differentiable functions.* Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[Sti09] S Stiller. Extending concepts of reliability. network creation games, real-time scheduling, and robust optimization, 2009.

[Van00] F. Vanderbeck. On dantzig-wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Operations Research*, pages 111–128, 2000.

A. Explanation of Techniques Used

A.1. Basic Problem

Many problems from practice have a basic, general form. When we take our repair facility example, we can see that we have a certain number of choices. In most formal ways to express a problem, we use variables to model choices. Most models also incorporate constraints. If we consider the simple form of our repair facility problem, we can make a yes/no choice for each repair, under the constraint that the amount of time the repairs we answer with a yes take to execute don't exceed 168 hours (one week). If we have binary variables x_i such that i is an index on our repairs, and we have constants a_i for the amount of time such a repair takes (in hours) and c_i for the profit we gain when we execute repair i, we can create a simple formulation of the problem in a formal way:

$$\begin{array}{lll} \max & \sum_{i \in I} c_i x_i & \\ \text{s.t.} & \sum_{i \in I} a_i x_i & \leq 168 \end{array}$$

This formulation is quite simple to read: the first line presents us with some objective. In this case it reads: maximize the amount of profit. The second line states the objective is subject to (s.t.) the major constraint: the amount of time necessary to execute our repair should not exceed 168 hours (one week). The last line states that our choices are binary (i.e. yes/no choices).

This formulation is an example of an Integer Linear Programming formulation (in short: ILP formulation). Many problems can be expressed using such a formulation. In general such problems have the following general form:

$$\begin{array}{lll} \max \text{ or } (\min) & cx & \\ \text{s.t.} & Ax & \leq b \\ & x \geq 0 & (\text{or } x_i \in \mathbb{N}, \text{ or } x_i \in \{0,1\}, \forall i) \end{array}$$

Here x is a vector of variables, c is a vector of profits or costs of the variables, A is the constraint matrix and b is the vector with the right hand sides of the constraint matrix.

Each variable can have its own upper and lower-bound. A variable can also have the constraint that it needs to be integral, or binary (either be 0 or 1). Using additional variables we can also create constraints of the form $Ax \geq b$ and $Ax \leq b$.

In many cases it is an option to ignore the integrality constraints on the variables (thus allowing fractional values). While this generally gives an infeasible solution to the original problem, it does give a valid upper- or lower-bound on the solution value of the original problem. Such a upper- or lower-bound can be used to determine the quality of your current solution (if a solution to a maximization problem is very close the value of the upper-bound, you know it can't be improved much more) and to prune the tree of a branch-and-bound algorithm as good as possible. A solution to an Integer Linear Program where the integrality constraints are ignored is called the LP-relaxation.

A.2. Column Generation

The column generation technique is a linear programming technique where not all variables are considered in the first place, but are added on the fly when they can improve the current solution. Suppose we have a standard linear programming problem of the form:

$$\begin{aligned} \max \quad & cx \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \end{aligned}$$

Now suppose we have a lot of variables, while a solution to the problem only has a few variables set to a value that is not zero.

The simplex algorithm is mainly concerned with jumping between extreme points in the convex space described by the constraint matrix. Given the system $Ax = b$, the Non-Basis matrix N is the sub-matrix of A that has the columns that correspond to the variables that have value zero in the current solution. The remaining columns from the matrix A are called the Basis-matrix B. The current extreme point is described by a inverted Basis-matrix, which the simplex algorithm keeps in memory.

All variables that are not currently in the Basis-matrix have a certain reduced cost. That cost can be calculated by taking the inverted Basis-matrix B^{-1} and the elements of cost vector the cost vector that correspond to the variables currently in the basis, known as c_B. Now if we take the shadow-price vector $c_B B^{-1}$ and take it's dot-product with the column vector A_i of a variable x_i that is not currently in the basis, we can calculate the price we have to pay to make room to bring it into the current Basis. These costs come from the notion that because of the constraints, when we give a new variable a value that is non-zero, the values of the other variables will have to change to keep the solution feasible. If we take the profit the variable generates, c_i, and subtract the costs it takes to make room for the variable, $C_B B^{-1} A_i$ also known as z_i, we get the so called *reduced costs* $c_i - z_i$. If we are maximizing our objective and this value is positive, our solution is improved if we bring the variable x_i into the current basis.

It is clear this trick can be used on variables that we know very well and are explicitly in the current LP-model. Another option is to begin with a very restricted model that has only a small amount of variables. Now we can solve that model and use the Basis of the optimal solution to the restricted problem to find a new variable that improves the solution if we add it to the restricted model. If we add that variable to the restricted model, we can use the old solution as a starting point to solve the new restricted model. This process can be repeated until the point where no variable can improve the solution, which implies the optimal solution has been found. A possible result is that only a few of all our variables from the original problem have been added to the restricted problem. If we had a *smart way* to find the next best variable to add to the restricted master problem, this could mean that the optimal solution has been found faster than in the case where the normal simplex algorithm would have been used.

This *smart way* to find the next best variable is the most important aspect of the column generation approach. If all variables are explicitly available but have some

structure that can be used to build a data structure that allows use to find the next best variable given the current value of $c_B B^{-1}$ in sublinear time, there is a considerable speedup. But if the number of variables is extremely large, it becomes desirable to find a way to prevent the exhaustive enumeration of all variables.

By using an implicit description of our variables, we can prevent such an exhaustive enumeration. While it isn't necessary, such an implicit description often has the form of another linear program. The solution to such a linear program gives us the next best variable, along with it's column in the constraint matrix of the restricted master problem and it's costs or profit. In many cases each extreme point from such a linear program is a potential variable in the master problem. Since the amount of extreme points is exponential in the size of the program it is clear to see that solving a linear program can be much faster than enumerating all possibilities.

The general name for the problem of finding a new variable with regard to some implicit description and the current values of $c_B B^{-1}$ is called the pricing problem. Because the values in $c_B B^{-1}$ are the link between the restricted master problem and the pricing problem, they are sometimes references by the name *shadow prices*.

Suppose we have a solution for the master problem. The shadow prices of the current solution are described by the vector σ. If we solve the pricing problem to find a new variable x_i, it should give us a new column in A_i and the costs of the new variable c_i. If we are maximizing our objective, we should check if the reduced costs, $c_i - \sigma A_i$ is positive. If we are minimizing our objective, we should check if it is negative.

During each iteration we should solve both the master and pricing problem. Solving the master problem each time is necessary to update the dual values, while solving the pricing problem is necessary because the dual values change. This way, we will increase the value of the master problem step by step. When no positive solution to the pricing problem is found, the process terminates. The algorithm is presented in pseudo-code in Figure 8.

One of the disadvantages of the column generation approach is that it is necessary to use linear programming without integer variables for the master problem. While it is possible to use the LP-relaxation to overcome this problem, this might lead to situations where the optimal solution is fractional (and thus not valid if your problem is an Integer Linear Problem). In such a case you have to be careful how to solve the situation. One approach is to solve the problem with your current set of columns as an integer linear program, hoping that you find a valid solution that has an objective value that is reasonably close to the objective value of the linear programming relaxation. If you do this and you are happy with the solution, you can stop. If you take the difference of the value of the integer solution and the value of the linear programming relaxation, also called the *integrality gap*, you have a measure of quality of the current solution. The smaller the integrality gap, the closer the current solution will be to the optimal solution.

Another approach is to use *branch and bound*, with the value of the LP-relaxation as a bound. Depending on the exact problem it can be hard to find a good branching rule. When branching on a regular LP-relaxation of a Integer Linear Program, you could choose a variable that has a fractional value in the current solution and branch two times: one time with it's upper-bound at the floor of the current value and one time with it's

Algorithm 8 Column Generation

```
Var Problem main
Var Problem pricing
Var Objective obj ← max ( or min )
main.solve()
Var Boolean improve ← True
while improve do
  improve ← False
  Var Vector σ ← main.getDuals()
  pricing.setDuals(σ)
  Var Vector A_i ← pricing.getColumn()
  Var Num c_i ← pricing.getCosts()
  pricing.solve()
  if (obj = max ∧ σ · A_i < c_i) ∨ (obj = min ∧ σ · A_i > c_i) then
    improve ← True
    main.addColumn(A_i, c_i)
    main.solve()
  end if
end while
```

lower-bound to the ceiling of it's current value.

With the column generation approach it is much harder to branch on a variable, since you generate variables all the time. In many cases, branching means adding or altering constraints, you can also branch on some variable in your pricing problem. When you alter your constraints, you introduce new dual values that need to be considered by your pricing problems. When you branch on a variable in your pricing problem, the result can be that some columns in your current solution become forbidden, which can leave you with an infeasible master problem. In general branching has an impact on your dual values that you need to consider. It also implies that it is important to generate new columns after branching, to make sure your bound remains tight, which is essential for the bound part of the branch and bound algorithm. This technique, where branch and bound is combined with column generation is called *branch and price*.

The major advantage of the column generation approach is the flexibility the pricing problem gives you. As long as it gives you new columns that improve your current solution, you can solve the pricing problem with any algorithm you want. This also means that you could put constraints that are hard to express in a linear fashion into the pricing problem. It also means that in a lot of situations you can use alternative algorithms, like shortest-path, network flow, or specialized dynamic programming algorithms. You are even allowed to use approximation or local search algorithms. You can even go as far as to use multiple algorithms: use a fast approximation algorithm to generate columns as long as possible and only switch to a slower exact algorithm when the approximation algorithm gives no improvement. In that way you still know your final solution to the master problem is optimal, while you might gain a huge speed boost because the majority

of your columns get generated by a much faster algorithm.

Column generation gives us the possibility to create decomposition algorithms. Without column generation it would be pointless to represent each possible sub-solution by a single variable. With column generation, such representations become meaningful, since the variables are considered implicitly instead of explicitly.

A.3. Dantzig-Wolfe Decomposition

Now suppose that we have some variable vector $x = x_1, x_2 \dots x_n$, and a *nice* partitioning of those variables into disjoint sets $x^1, x^2 \dots x^k$, etc. The definition of *nice* is omitted, since no fool proof method exists to do this. The rule of thumb is to choose a partitioning is such a way that the problem is decomposed in nice sub-problems. Suppose that we have some constraints that are defined on the variables of some disjoint set and have a few constraints that are defined on variables from different sets. In other words: our constraint matrix A contains (non-zero) sub-matrices D_i and F_i for $1 \leq i \leq k$ and b is split into sub-vectors b^i for $0 \leq i \leq k$, such that we have the following structure:

$$\left(Ax = \begin{array}{cccc} D_1x^1 & +D_2x^2 & \dots & +D_kx^k \\ F_1x^1 & +0 & \dots & +0 \\ 0 & +F_2x^2 & \dots & +0 \\ \vdots & \vdots & \ddots & +0 \\ 0 & +0 & +0 & +F_kx^k \end{array} \right) = \left(\begin{array}{c} b^0 \\ b^1 \\ b^2 \\ \vdots \\ b^k \end{array} = b \right)$$

Now consider the n-dimensional space described by the variables in x and the hyperplanes trough this space described by the matrix A. Now consider the d^i-dimensional spaces described by each variable set x^i for $1 \leq i \leq k$, such that $d^i = |x^i|$. A point in such a d^i-dimensional space will project to an affine subspace in the space described by x. If we take a point for each space described by an x^i set and project all these points to subspaces in our n-dimensional space, these subspaces will intersect in a single point. Thus, a point in the n-dimensional space can be described by a point in each of the d^i-dimensional spaces in our partitioning of x.

Example Suppose we have three dimensional space, with variables x, y and z. In a three dimensional space, both a ray and a plane are possible subspaces. Suppose we partition these variables into two sets: $\{x, y\}$ and $\{z\}$. Suppose we pick a single point in the two dimensional space described by x and y. If we project this point onto our three dimensional space, we get a single ray that is parallel to the z-axis. Now suppose we also choose a single point in the one dimensional space described by z. If we project such a point onto our three dimensional space, we get a plane that is parallel to the plane described by both the x and y axes. Since the plane and the ray are not parallel to each other (the plane is not parallel to the z, while the ray is), there must be a single point where they intersect. Thus, a point from $\{x, y\}$ and a point in $\{z\}$ describe a point in $\{x, y, z\}$ by intersection.

This implies that each extreme point of the system $Ax = b$ can be described by a combination of points from each x^k space. As a matter of fact, the theory of the Dantzig Wolfe decomposition tells us that each extreme point of the system $Ax = b$ can be described by combining an extreme point from each $F_k x^k = b^k$ system.

The idea is to create a master problem that can have variables y_i^k for each extreme point in each disjoint set x^k, where y_i^k represents the ith extreme point x_i^k. Suppose that the vector c^k is a sub-vector of c, such that c^k contains the cost factors for the variables in x^k. Our master problem will have variables y_i^k and look like this:

$$
\begin{array}{rll}
\text{max or min} & \sum_k \sum_i (c^k x_i^k) y_i^k & \\
\text{s.t.} & \sum_k \sum_i (D_k x_i^k) y_i^k = b^0 & \text{duals: } \sigma_0 \\
& \sum_i y_i^k = 1 & \forall k \text{ dual: } \sigma_k \\
& y_i^k \geq 0 & \forall k, \forall i
\end{array}
$$

Since the optimal solution will only select a single extreme point from each of our disjoint sets and since enumerating all points takes up too much time, we will not consider all extreme points explicitly. By using column generation, we will generate extreme points that can possibly improve the current solution to a better one. This means we also need a pricing problem to find a new extreme point for some set x^k that improves the solution.

Since we look for k extreme points, we have k different pricing problems. The reduced costs of an extreme point from some set k are described by $c^k x^k - (D_k x_k)\sigma_0 - \sigma_k$. Using our reduced cost formulation, we describe the pricing problem of finding a new extreme point for a set k with only the variables in x^k as:

$$
\begin{array}{rl}
\text{max or min} & c^k x^k - \sigma_0 D_k x^k - \sigma_k \\
\text{s.t.} & F_k x^k = b^k \\
& x^k \geq 0
\end{array}
$$

This decomposition technique is applicable to any linear program with a partitioning of the variables that follows the given structure. It is even possible to use this approach partially on a program. In that case some of the original variables from the original problem are left alone in the master problem, while some other variables are replaced by new variables that represent extreme points, by moving the original variables into the pricing problem. Such an approach gives a partial Dantzig-Wolfe decomposition.

B. Implementation of Algorithms

B.1. Data Structures

B.1.1. Items, Scenarios, Problems and the LinkedKnapsack

The most basic data structures used in the algorithms, are the Item, the Scenario (including the WeightedScenario) and the LinkedKnapsack. The Item is a simple data structure that holds an integer defining it's size and a floating point number defining it's profit. A basic scenario simply has an integer defining it's size. The extension, a WeightedScenario, add an floating point number defining it's weight.

The Problem class has a collection of Items and a collection of scenarios. It caches a simple copy of the scenario with the highest value, so this one is easily accessible. Two helper methods to read and write problems from disk were written, to make life a lot easier.

The LinkedKnapsack is the easiest implementation of an Knapsack and is very similar to the well known single linked list. A LinkedKnapsack is simply a node that holds an Item, it's size, it's value and a pointer to a parent LinkedKnapsack. A LinkedKnapsack k can easily be extended with an Item i: a new LinkedKnapsack k' is created, it's parent becomes k, it's value becomes $value_k + c_i$, it's size becomes $size_k + a_i$ and it's Item becomes i. Note that both k and k' will remain in memory. When we want to iterate over all Items in a LinkedKnapsack, we can simply follow the pointers to the parents, until we find a node without a parent. Note that it is not possible to remove an Item from a LinkedKnapsack - to do so you will either need to create a new LinkedKnapsack structure from scratch, or you will need to backtrack to the node where the Item was added, and extend a new path from it's parent.

B.1.2. Recovery Knapsack Implementations

Since the value of the LinkedKnapsack is based on a single scenario and only contains a single set of Items, we need an extension that is capable to represent a solution to our problem. For this purpose the solution needs to know the exact set of items that will be used for each scenario, as well as the weighted value of the solution. There are a few ways to do this and the first implementation was the RecoveryKnapsack.

The RecoveryKnapsack is similar to the LinkedKnapsack. A LinkedKnapsack holds an item, it's size, the sum of the values of all the items in it, a recovery vector and a pointer to a set of scenarios. When an RecoveryKnapsack is extended with an item, the size and profit of the item are added to the size and the value of the previous knapsack. The recovery vector is created by using the dynamic programming recurrence for recovery on the recovery vector of the parent node. When the true value of the LinkedRecovery knapsack is requested, it can be calculated on the fly by iterating trough the set of scenarios and subtracting $w_s r_{size-b_s}$ from the total value, where r_{size-b_s} is the recovery value reported by the recovery vector. If $k \leq 0$ then $r_k = 0$. Otherwise it will have the summed value of the cheapest way to remove items of at least size k.

Like the LinkedKnapsack, the RecoveryKnapsack has no operation to remove an

item. Since the local search algorithms need this functionality, we need to create an implementation that makes this possible. The first idea is to take the RecoveryKnapsack and add an algorithm to it that does backtracking trough the tree to the node where the item was added and extend the parent of that node with all the items visited during the backtracking process. In the worst case this means that the entire knapsack will have to be recalculated, in case the root from the tree will be removed, but it gets the job done. This implementation is called MutableRecoveryKnapsack.

Another implementation which is important for certain algorithms, like the Exact Dynamic Program, is the PartitionRecoveryKnapsack. Instead of holding a single item-set and calculating the recovery for each scenario, the PartitionRecoveryKnapsack holds the explicit set of items for each scenario. When an item is added, you need to specify to which scenario item-sets the item should be added. Due to the nature of the problem it is forbidden to add an item to a item-set if it is not also added or already in the item-set for the main scenario. Removing an item can also be done easily, but removing an item from the main scenario means it also has to be removed from all sub scenarios. During the process of updating all item-sets, the sizes and values for each of these sets can be updated efficiently.

B.1.3. DPHashTable

Since some of the Dynamic Programming algorithms need a large table to memorize the results and since these tables have an arbitrary number of dimensions, a special data structure was created to facilitate these needs.

The DPHashTable is a structure of HashMaps. When the table is initialized, the number of dimensions d must be specified. A store operation is supported, that accepts an object and exactly d integers. For each integer, it is checked if the current HashMap contains a key equal to the integer. If not, a new empty HashMap is store with the integer as a key and the new Hashmap becomes the current Hashmap. If the HashMap contains the key, the new current Hashmap is retrieved from the current one. If the new integer is reached, the object is stored in the current hashmap.

The retrieve operation also takes exactly d integers and works in a similar fashion: the HashMap structure is recursively visited with the integers as keys. If an object is found at the end, the object is returned.

There is also a getFullSet operation, that goes in recursion trough the entire data structure and returns all objects stored in the structure.

B.2. Algorithms

B.2.1. Separate Recovery Branch and Price

The Separate Recovery Branch and Price algorithm contains three important parts: the implementation of the linear programming model of the main model, the dynamic program that solves the pricing problem and the branching algorithm.

The linear programming implementation is a very straightforward implementation of the model presented in section 4.1.1 using CPLEX. When a problem is passed to the

implementation it builds the LP-model in CPLEX, keeping some administration as to which constraints map to which scenario or item. This is primarily done with HashMaps. To make the starting solution feasible, the empty knapsack is added as a variable for each solution.

Since the column generation can be made faster by giving it a good starting solution, the implementation has also the option to generate a better starting solution. If this option is enabled, the implementation will call the HillClimbing algorithm 10 times and add the best solution to the model.

Besides the LP-model we have the Column Generation algorithm. This algorithm iterates over all scenarios and creates a new item-set based on the shadow prices for the current solution, only containing the items that still have a positive profit. It also creates a mapping to be able to translate a solution to a pricing problem to the real problem. It then calls the basic Dynamic Programming algorithm for the generated item-set and the bound of the current scenario. If there is a solution, it is added as an column and the model is solved again. If a complete iteration of all the scenarios fails to generate a new column, the process of generating columns is stopped.

This is when the branching algorithm needs to do its job. After the columns are generated, it checks if the solution to the model is worse than the current lower bound: if it is, we can stop examining the current branch. After this it checks if the current integer solution for the model is better than the current lower-bound. If it is, the solution becomes the new lower-bound. The next step is to check if the current solution to the model is an integer solution: if this is the case, we can stop examining the current branch.

If neither of this is the case, we will have to branch. We have to select an item to branch on. The implementation can be configured to choose either an item that is fractional in the current solution or an item that is not yet fixed by branching. In both cases it is possible we can choose multiple items. We can configure the implementation to choose the item with either the largest or smallest ratio, size or profit. After the item is selected we need to branch two times: one time we need to fix the item in the main solution for the knapsack, branch and unfix the item. The other time we need to forbid the item in the solution, branch and re-allow the item in the solution. The implementation has a configuration option which one is done first.

Fixing or forbidding an item can be done by adding a proper constraint to the model. These steps can be undone by removing the constraint that was added. When a constraint is added or removed this changes the pricing problem, so this means that the column generation algorithm must be able to cope with this kind of behavior.

B.2.2. Combined Recovery Branch and Price

The Combined Recovery Branch and Price algorithm has a lot in common with the Separate Recovery Branch and Price when we consider the implementation. The implementation of the model is done in the same way and the branching algorithm is also very similar. The column generation phase has a few additional features in it's implementation, the most important is the choice to generate all sister-columns when a new column is found. Since a column represents the solution for a main knapsack and has the value

for a single scenario, we can also add columns for the other scenarios by calculating the proper recovery values for all these scenarios. If we do this, we are sure that the new column can be selected, since the same solution can be chosen for all the other scenarios, which at least gives the model a possibility to choose the column with a value larger than 0. Besides this, stabilized column generation [dMVDH99] was also implemented, but since adding all columns also seems to speed up the algorithm, stabilized column generation wasn't used during the experiments.

B.2.3. Branch and Bound

The Branch and Bound algorithm we discussed in Section 5.3.6 is a basic branching algorithm, programmed in a recursive manner. When the algorithm is initialized, the items are sorted in a descending or ascending order by ratio, size or profit. When the algorithm is executed, it calculates an upper-bound for the current solution. If the current lower-bound is equal or higher than the upper-bound, the current execution is stopped and the recursion goes back a level.

Two relaxations for an upper-bound have been implemented. The simple one takes all the items that have not been forbidden trough branching and calculates the LP-relaxation for the problem with only these items. The other one generates a knapsack problem for each scenario: the main scenario gets a knapsack problem with only items that have not been decided upon and the remaining space in the main knapsack. For the other scenarios we calculate a single knapsack Dynamic Programming vector and take the best values for each scenario. The weighted sum of these calculations is the Recovery relaxation discussed in Section 5.4.6.

If the upper-bound is greater than the lower-bound, the algorithm continues by taking the next item from the list. Depending on the order specified in the configuration it forbids the item and goes into recursion, or it creates a new RecoveryKnapsack node by adding the item and going in recursion.

B.2.4. Exact Dynamic Programming

The Exact Dynamic Programming algorithm is based on the Dynamic Programming recurrence from section 5.3.5. Since it is not trivial to implement an array of an arbitrary number of dimensions, the DPHashTable structure was used to represent a single table for each item iteration. When an item is passed to the algorithm, a new DPHashTable is created and an iteration on all PartionRecoveryKnapsacks from the previous table is started. The recursion is used on all these PartitionRecoveryKnapsacks and the result are added to the new table. At the end of the iteration, the previous table makes way for the new table.

B.2.5. Iterative Dynamic Programming

The Iterative Dynamic Programming is an algorithm that uses the Dynamic Programming recurrence from 5.4.1. While this recurrence is invalid for solving the problem to optimality, it gives an approximation of the solution value. Besides that, the algorithm can give an

optimal solution if the order of the items is correct: if you iterate trough the items that are in the optimal solution before you iterate trough the other items, the resulting table will contain the optimal solution, since it can't be dominated by other solutions.

Because the order is important, we will consider the three ways to sort the items. After each iteration trough all items, we will reverse the order of the items, so different orderings are tried and used.

Another way to use the algorithm is to shuffle all the items after each iteration in a random fashion, hoping that a good order is reached during one of the iterations, which will result in a nice solution.

B.2.6. Hillclimbing

The Hillclimbing algorithms uses iterative improvement. The neighborhood of a solution are all solutions to which an item is added (if the capacity constraint allows it), or in which an unused item and a used item are swapped (if the new item has a higher profit and the capacity constraint allows it). It is possible to ignore the swaps, but in that case the algorithm just greedily adds the items with the highest values.

To make the Hillclimbing perform better, we add an random restart feature to it. Random restart means that a random starting solution is used, which is improved afterward. The best solution of this process will be the ultimate solution to the Hillclimbing process. To make swaps possible, the Hillclimbing algorithm makes use of the MutableRecoveryKnapsack implementation, where a swaps is just a remove followed by an add operation.

B.2.7. Simulated Annealing

The Simulated Annealing algorithm uses three types of operations: adding items, removing items and swapping items. Since Simulated Annealing allows decreasing the value of a solution and removing an item from a MutableRecoverKnapsack can take a lot of time, it is wise to take the random threshold, estimate a lower bound on the decrease of the solution value and decide if the new solution might be accepted, before calculating the real value of the solution. To methods were implemented to get a random neighbor: one method is to generate all possible transformations on the current solution and pick one add random. The other method is a Las Vegas algorithm [Bab79] that creates random transformations until one that is feasible feasible is found.

Two possible cooling schemes were implemented: a linear cooling scheme and an exponential cooling scheme [RN03]. The exponential cooling scheme multiplies the current amount of energy by a certain value, probably close to 1. The linear cooling scheme subtracts a certain value, probably close to 0, from the current energy. There is an initial amount of energy and an amount of energy that when reached stops the algorithm. Besides this, the algorithm allows the specification of a number of steps before which cooling is done and the algorithm also allows the specification of a maximum number of steps that can be done without improving the best solution found.

To improve the chance to reach local optima, an additional operation was implemented,

which was given a chance parameter. If this operation was chosen, the algorithm would perform an iterative improvement on the current solution, to enforce that the current solution becomes a global optimum. This way the algorithm could be tweaked to find better solutions in certain cases.

B.2.8. Tabu Search

The Tabu-Search algorithm takes two arguments: the length of the recovery list k and a maximum number of steps. The operations Tabu Search can perform on the current solution are the addition of an item, the removal of an item and possibly swapping an used item for an unused item. Swapping can be turned on or off. The usage of the maximum number of steps can also be selected: it can be the number of steps the algorithm does before termination, or it can be number of steps since the last improvement.

The current solution of the Tabu Search algorithm is a single MutableRecoveryKnapsack. The tabuList contains simple HashSets of items, which makes comparing solutions more efficient.

C. Detailed Proofs

C.1. Bellman Recurrence

We will show that the following variant of the Bellman-Recurrence is correct.

$$
\begin{aligned}
A(i,0) &= \quad 0 \\
A(0,b) &= \quad -\infty \qquad \text{(for } b \neq 0) \\
A(i,b) &= \quad \max\{A(i-1,b), A(i-1,b-a_i)+c_i\}
\end{aligned}
$$

Proof. We want to show that for a certain index i on the item-set I, $A(i,b)$ gives the optimal value for item-set size b. We will do this using mathematical induction.

Base case For $i = 1$, we only consider one item in I. The only possible sets are $\emptyset$ at $b = 0$ and $\{1\}$ at $b = a_1$. These two cases are represented by the recurrence for $i = 1$ and all other cases are represented by a value of $-\infty$, which is correct.

Inductive Step For a certain i, we assume that for all possible values b the value of $A(i-1,b)$ is optimal. Under this assumption we will show that for all possible values b the value of $A(i,b)$ will also be optimal.

We start with the assumption that for some value of b, $A(i,b)$ is not equal to the optimal value z_b^* for the items 1 to i. We know that $b > 0$, because for $b = 0$ the only possible value is 0, so $A(i,0)$ will be optimal according to the recurrence. Since $b > 0$, we know that $A(i,b)$ is the maximum of $A(i-1,b)$ and $A(i-1,b-a_i)+c_i$. We now have two cases:

1. Item i is not in the item-set that corresponds with z_b^*. In this case z_b^* corresponds to an item-set which only contains items in the range of 1 to $i-1$. Since $A(i-1,b)$ was considered for the value of $A(i,b)$, we know that $z_b^* > A(i-1,b)$. However, we also know that z_b^* is a valid value for $A(i-1,b)$, so this contradicts our assumption that for all b we have that $A(i-1,b)$ is optimal.
2. Item i is in the item-set that corresponds with z_b^*. If this is the case, we can remove i from the item-set. We now get an item-set that contains only items from 1 to $i-1$ and has value $z_b^* - c_i$, since item i was removed. We also know that $A(i,b) - c_i < z_b^* - c_i$. However, $A(i-1,b-a_i)+c_i$ was considered for the value of $A(i,b)$, so it must be the case that $A(i-1,b-a_i) < z_b^*$. This means that we can construct an item-set from z_b^* that is better than the item-set that corresponds to $A(i-1,b-a_i)$, which is a contradiction against our assumption that for all b the value $A(i-1,b)$ is optimal.

We now have derived that the values for $i = 1$ are correct, and have also shown that if the values for i are correct if the values for $i-1$ are correct. Thus we have completed the inductive proof. □

C.2. Balanced SSP Algorithm Completeness

Let us consider the correctness of the algorithm from Section 5.2.2. Suppose there is some item-set $I^* \subset I$ that is the optimal solution. Let us define a current $\hat{I}$ that corresponds to the split solution $\hat{x}$. If $\hat{I}$ is equal to I^*, we are done. Now, as long as $\hat{I}$ is not equal to I^*, there are two possibilities: $\sum_{i \in \hat{I}} a_i \leq b$ or $\sum_{i \in \hat{I}} a_i > b$. In the first case, we add the item with the lowest index that is in I', but not in $\hat{I}$. In the second case, we remove the item with the highest index that is in $\hat{I}$ but not in I'. If the operation for the first case cannot be performed, we have that $\hat{I}$ contains items not in I^* (since it is not equal to I^*), so I^* must be a strict subset of $\hat{I}$. Since $\hat{I}$ is feasible, this situation contradicts the assumption that I' is optimal, because $\hat{I}$ is larger and feasible. If the operation for the second operation cannot be performed, we have that $\hat{I}$ must be a strict subset of I^*. But since $\hat{I}$ is already infeasible, so must be the case for I', which again contradicts the assumption that it is optimal. Since the state I^* must be reachable from state $\hat{I}$ and since the added items must have ascending indices, while the removed items have descending indices, the algorithm has the possibility to find item-set I^*. While item-set I^* can be dominated by an item-set I', but this can only happen if I^* and I' have equal size. Since we consider the subset sum problem, item-set I' is just as good as item-set I^*.

C.3. Lagrangian Relaxation of RKP-R

In Section 5.4.4, we discussed an alter Let us consider the Langragean Dual Problem for this alternative formulation of the RKP-R, where the capacity constraints are relaxed. Our Lagrangean Dual Problem $\min z(\lambda)$ is the following

$$\begin{aligned} z(\lambda) = \max & \sum_{i \in I} (c_i - \sum_{s \in S'} \lambda_s a_i) x_i + \sum_{i \in I} \sum_{s \in S} (\lambda_s a_i - p_s c_i) \bar{y}_i^s + \sum_{s \in S'} \lambda_s b_s \\ \text{s.t. } & \bar{y}_i^s \leq x_i, \forall s \in S, \forall i \in I \end{aligned}$$

We will start by removing the $\bar{y}_i^s$ variables from the problem.

To make our life easier we will define helper sets $\hat{S}_i$ for some $i \in I$ and $\hat{I}_s$ for some $s \in S$ that depend on the current Lagrangian multipliers. We will use these sets to make a distinction between variables that will be put to 1 and variables that will be put to 0.

Helper Sets

$$\begin{aligned} s \in \hat{S}_i \leftrightarrow \lambda_s a_i > p_s c_i \quad & \text{for some } i \in I \\ i \in \hat{I}_s \leftrightarrow \lambda_s a_i > p_s c_i \quad & \text{for some } s \in S \end{aligned}$$

We begin with a simple observation: since the knapsack constraints are gone we are free to put all our x_i variables to 1 and put all $\bar{y}_i^s$ variables to 1 if the corresponding x_i variable is set to 1. Doing this might not result in the optimal value, because setting certain variables to 1 might have a negative contribution because of the Lagrangian

multipliers. Let us first consider the $\bar{y}_i^s$ variables:

$$\bar{y}_i^s = 1 \quad \text{is profitable iff} \quad x_i = 1 \wedge \lambda_s a_i > p_s c_i$$
$$\text{or}$$
$$\bar{y}_i^s = 1 \quad \text{is profitable iff} \quad x_i = 1 \wedge s \in \hat{S}_i$$

Using this property we can rewrite our objective function to a tighter formulation, by tightening the scope of the summation over the $\bar{y}_i^s$ by removing $\bar{y}_i^s$ variables from it that will be zero.

$$z = \sum_{i \in I} (c_i - \sum_{s \in S'} \lambda_s a_i) x_i + \sum_{i \in I} \sum_{s \in \hat{S}_i} (\lambda_s a_i - p_s c_i) \bar{y}_i^s + \sum_{s \in S'} \lambda_s b_s$$

Now let us consider the x_i variables. Whatever happens, we are allowed to set a variable x_i to 1, so the only question here is if this is profitable. Putting x_i to 1 adds $c_i - \sum_{s \in S'} \lambda_s a_i$ to the objective function. If this expression has a positive value, we know it is a good idea to put x_i to 1. However, if this expression has a negative value it might be a good idea to put it to 1 anyway if this allows us to set enough $\bar{y}_i^s$ variables to 1 to compensate for the loss. Using a helper set $\hat{S}_i$ we can determine which $\bar{y}_i^s$ variables will correspond to a scenario that will give a positive contribution. We now get the property

$$x_i = 1 \quad \text{is profitable iff} \quad c_i - \sum_{s \in S'} \lambda_s a_i + \sum_{s \in \hat{S}_i} (\lambda_s a_i - p_s c_i) > 0$$

Using the expression $\sum_{s \in \hat{S}_i} (\lambda_s a_i - p_s c_i)$, we express the total value of the $\bar{y}_i^s$ variables that are profitable given the corresponding x_i variable is set to 1. Besides giving a powerful tool for determining if a x_i variable should be set to 1, it also gives us the possibility to remove the $\bar{y}_i^s$ variables from the problem entirely. Doing this we get the objective function

$$z = \sum_{i \in I} (c_i - \sum_{s \in S'} \lambda_s a_i + \sum_{s \in \hat{S}_i} (\lambda_s a_i - p_s c_i)) x_i + \sum_{s \in S'} \lambda_s b_s$$

Since the only remaining variables in the problem are the x_i variables, we isolate the effective price c_i' for certain variable x_i

$$c_i' = \quad (c_i - \textstyle\sum_{s \in S'} \lambda_s a_i) + \sum_{s \in \hat{S}_i} (\lambda_s a_i - p_s c_i)$$
$$\text{extended to}$$
$$c_i' = \quad c_i - \textstyle\sum_{s \in S'} \lambda_s a_i + \sum_{s \in \hat{s}_i} \lambda_s a_i - \sum_{s \in \hat{S}_i} p_s c_i$$
$$\text{simplified to}$$
$$c_i' = \quad c_i - \textstyle\sum_{s \in \hat{S}_i^{-1}} \lambda_s a_i - \sum_{s \in \hat{S}_i} p_s c_i$$

The nice thing we see is that we have a summation over $\hat{S}_i$ and one over $\hat{S}_i^{-1}$, which implies that for a single item i each scenario s either contributes $p_s c_i$ or $\lambda_s a_i$, depending if it is or is not in the set $\hat{S}_i$. Now that we have reduced the Lagrangian relaxation of

our problem to a much simpler form, we will take a look at the Lagrangian dual problem, which states that we should choose our Lagrangian multipliers in such a fashion that the value of the solution of the relaxed problem is minimized. To achieve this we will take a look at the impact changing a certain multiplier has on the value of the relaxed problem. Let us first consider a case where we change a certain multiplier λ_s by $\Delta\lambda_s$ in such a way that $\hat{I}_s$ remains unchanged. Doing this we can clearly see that we only have to recount all factors that contain λ_s. Doing this we derive the effect on the objective function Δz

$$\Delta z = \Delta\lambda_s b_s - \sum_{i \in \hat{I}_s^{-1}} \Delta\lambda_s a_i \quad \text{in case } \hat{I}_s \text{ remains the same}$$

Taking a look at the definition of c_i', we can see that changing a certain multiplier λ_s only changes the c_i' of the items with an index i such that $s \in \hat{S}_i^{-1}$, since the items without such an index contribute $-p_s c_i$ to the objective, which is not dependent on λ_s.

Let us now consider the case where we change a certain λ_s by $\Delta\lambda_s$ in such a way that only a single item j is transferred from $\hat{I}_s^{-1}$ to $\hat{I}_s$. In this case we have the same effect on all λ_s factors, but we also have some item that loses its λ_s factorization and now counts for $p_s c_i$.

$$\begin{aligned} \Delta z = \quad & \Delta\lambda_s b_s - \sum_{i \in \hat{I}_s^{-1}} \Delta\lambda_s a_i + (\lambda_s - \Delta\lambda_s) a_i - p_s c_i \\ & \text{in case only item } j \text{ is transferred from } \hat{I}_s^{-1} \text{ to } \hat{I}_s \end{aligned}$$

In our next step we will consider the set $\Delta\hat{I}_s^{-1}$ given some $\Delta\lambda_s$ applied to some λ_s as the set that contains all items that are transferred from $\hat{I}_s^{-1}$ to $\hat{I}_s$. Using this definition we finally get the definite relation between Δz and $\Delta\lambda_s$.

$$\begin{aligned} \Delta z = \quad & \lambda_s b_s - \sum_{i \in \hat{I}_s^{-1}} \Delta\lambda_s a_i + \sum_{i \in \Delta\hat{I}_s^{-1}} (\lambda_s - \Delta\lambda_s) a_i - \sum_{i \in \Delta\hat{I}_s^{-1}} p_s c_i \\ & \text{where } \Delta\hat{I}_s^{-1} \text{ contains the items transferred from } \hat{I}_s^{-1} \text{ to } \hat{I}_s \end{aligned}$$

Now that we have Δz, we can split our objective function z into components that depend on a single multiplier λ_s

$$\begin{aligned} z = \quad & \sum_{s \in S'} z_s \\ & \text{where} \\ z_s = \quad & \lambda_s b_s - \sum_{i \in \hat{I}_s^{-1}} \lambda_s a_i - \sum_{i \in \hat{I}_s} p_s c_i \end{aligned}$$

Because the Lagrangian dual problem tells us to minimize the value and we have a closed formula for the contribution of a certain multiplier λ_s, we only have to choose each λ_s in such a fashion that the corresponding z_s is minimized. When we look at the helper sets, we see that λ_s divides $\hat{I}_s^{-1}$ and $\hat{I}_s$ and also orders the items according to their ratio's $\frac{c_i}{a_i}$. For a certain s, the contribution of a single item i lies in the range $\{0... - p_s c_i\}$. If we have $\lambda_s > \frac{p_s c_i}{a_i}$, item i will be in $\hat{I}_s$ and thus contribute $-p_s c_i$. However if $\lambda_s < \frac{p_s c_i}{a_i}$, item i will contribute $-\lambda_s a_i$, which is of course smaller than $p_s c_i$. Using this we can express the condition that states when raising λ_s will lower z_s.

$$\uparrow \lambda_s \text{ leads to } \downarrow z_s \quad \text{while} \quad \lambda_s b_s \leq \sum_{i \in \hat{I}_s^{-1}} \lambda_s a_i$$

Theorem C.1. *To minimize the value of the Lagrangian dual problem, we should set multiplier λ_s to $p_s \frac{c_{\overline{i_s}}}{a_{\overline{i_s}}}$ where $\overline{i_s}$ is the split item for b_s.*

Proof. Since the items in $\hat{I}_s$ and $\hat{I}_s^{-1}$ are divided according to their ratio's and the condition states that we should fill $\hat{I}_s^{-1}$ to the point where the sum of the weights of the items exceeds b_s. That means we should take the item that fills the item-set of scenario s up to the value b_s as the dividing item for $\hat{I}_s$ and $\hat{I}_s^{-1}$. The proper value for λ_s thus becomes $p_s \frac{c_{\hat{i_s}}}{a_{\hat{i_s}}}$ where $\hat{i_s}$ is the split item for b_s. □

C.4. A straightforward reduction of 3-Partition to RKP-R fails

Let us consider the question whether the RKP-R is NP-hard in the strong or in the weak sense. Since it is a generalization of the regular KP, it is clear that it is at least NP-hard in the weak sense.

The article [GJ78] states that we can prove a problem to be NP-hard in the strong sense if we are able to find a polynomial time reduction from a problem that is NP-hard in the strong sense. A classic example of such a problem is 3-Partition, which states:

3-Partition *Input:* Given $3N$ integers $a_1, \ldots, a_{3N}$ that sum up to $B = \sum_{i=1}^{3N} a_i$.
Question: Is there a partition into triplets such that each integer $i \in 1, \ldots, 3N$ is in exactly one triplet and for each triplet (a_i, a_j, a_k) we have $a_i + a_j + a_k = \frac{B}{N}$?

Now the straightforward way to do this is by using the RKP-R as a Subset Sum problem by fixing the ratio of our items to 1. We consider all integers from our 3-Partition problem as items. Additionally we create N scenarios, such that for each scenario s we have $b_s = (|S| - s)\frac{B}{N}$. Now in the case of $N = 2$, we know there is a solution to 3-Partition *if and only if* the RKP-R gives an exact filling for both scenarios, due to the proof from Section 5.3.5.

However, for $N > 2$, this approach fails. Consider the following example: we have $N = 3$ and our integers are $\{6, 4, 6, 4, 6, 4, 6, 4, 5\}$. This sums nicely to 45, so we have $\frac{B}{N} = 15$. We create scenarios $b_0 = 45$, $b_1 = 30$ and $b_2 = 15$ for the RKP-R. This gives us an exact solution: all items as an initial solution, the item-set $\{6, 4, 6, 4, 6, 4\}$ for scenario 1 and the item-set $\{6, 4, 5\}$ for scenario 2. This contradicts the fact that we have no solution to 3-Partition, due to the single odd number 5 and the requirement of 3 partitions with the requirement to have an odd sum.