Chapter 1

Introduction

Robust optimization deals with optimization problems where the underlying data making up the objective function and the constraints are, at least partially, subject to uncertainty. Ways how to deal with uncertainty are manifold and depend largely on assumptions about the nature of the uncertainty and which part of the problem it affects. The uncertainty may be known, for example, as a probability distribution or simply as a set of possible values, which may then be finite, discrete or continuous. It may affect different parts of the problem: the objective function or the constraints, partially or fully.

The nature of the uncertainty and where it affects the problem data is determined by the source of the uncertainty. Sources of uncertainty in mathematical optimization problems are just as diversified as the real-world problems they model. If data is measured, it may be subject to measurement error. Data may be influenced by previously taken and possibly faulty decisions. Data may also depend on decisions to be taken by another entity which may be influenced to assist your needs, or by an adversary who wants to see your plans foiled. Information may be imprecise, forged, scarce or simply not yet available at the point when a decision has to be taken with the help of a mathematical programming model.

Data uncertainty can often be regarded as either a restriction or a chance, depending on whether one assumes an optimistic or a pessimistic point of view. The optimist hopes that he can eventually choose a specific instantiation of the data that will serve him best. The pessimist, however, assumes that all possible instantiations of data may occur without him having any means of influence and that he therefore has to prepare for the worst. These two ways of regarding data uncertainty lead to two classes of mathematical programming models, which are different but nonetheless related, as we will see later. Speaking of robust optimization, one usually refers to a pessimistic interpretation of the uncertainty. The data is regarded as being perturbed by some adversary entity, and one

tries to be protected against these perturbations.

Throughout this thesis, we choose a deterministic viewpoint rather than a probabilistic one; uncertainty is in the form of sets of possible data instantiations. In the course of this dissertation, these sets get increasingly more general: We start with sets with simple structure, later we look at general polyhedra, then at ellipsoidal sets, and we finally discuss discrete uncertainties defined by the integer points of a polyhedron. We assume that the uncertainty is provided to us, and we do not consider the process of retrieving appropriate uncertainty sets for specific applications. As to optimistic and pessimistic views on uncertainty, both are covered but with a stronger focus on the pessimistic case, i.e., we mainly consider robust optimization.

Chapter 2 introduces notation and preliminaries helpful to read this thesis, followed in Chapter 3 by a short review of what has been done in robust optimization and what provided a basis for the topics covered by this thesis. What can be found in literature about robust optimization focusses mainly on tractability. Either a simple problem, like a linear program, is taken and perturbed by quite general uncertainties, like polyhedra or ellipsoids; or, the underlying problem is harder, like a 0-1 program, but the uncertainties have a very simple structure that allows the reduction of the problem to a series of non-robust problems of the same type as the underlying problem. What has not been done yet is to look at problems that are generally considered intractable by themselves, like MIPs, but that are additionally subject to uncertainty of a general type. Subsequent chapters are dedicated to research this direction.

In Chapter 4, we investigate a generalization of a model from the literature for uncertainty in the costs of a specific, simply structured type that allows the control of the amount of robustness in a problem. In the literature, this is done by a single constraint restricting the number of simultaneously uncertain coefficients. In the generalization presented here, more than one constraint and general knapsack constraints with positive coefficients are allowed. The generalization thus offers much more flexible ways to impose control on the uncertainty.

We show that a robust 0-1 program with controlled uncertainty in the cost is solved by a finite number of non-robust 0-1 programs with the same set of feasible solutions. This number is bounded by a polynomial in the dimension of the problem as long as the number of knapsack constraints in the control is bounded by a polynomial as well. We further outline a subgradient method for the robust minimum-cost flow problem with controlled uncertainty in the cost that converges to the optimum while iteratively solving non-robust minimum-cost flow problems.

In the second major part of the thesis, Chapter 5, linear and mixed-integer programs under uncertainty are considered. Both the pessimistic view represented by robust linear and robust mixed-integer programs and the optimistic view represented by generalized linear and generalized mixed-integer programs are looked at. Reformulations of these problems with polyhedral uncertainty as non-robust linear or mixed-integer programs are reviewed. With the help of these, duality between robust and generalized linear programs, and some Farkas-like lemmas are derived. A solution method from the literature for generalized linear programs is presented. It is then apparent that robust and generalized linear programs are efficiently solvable.

We then primarily aim at deriving cutting planes for mixed-integer programs under uncertainty. We prove that lattice-free cuts for robust mixed-integer programs are generated by solving certain generalized linear programs. Also, lattice-free cuts for generalized mixed-integer programs are generated by solving robust linear programs. Finally, it is demonstrated how the results of this part are generalized to uncertainties described by convex conic sets.

The last part, Chapter 6, is dedicated to implementation and computation. The lattice free cuts for robust mixed-integer programs are compared to lattice-free cuts for the mixed-integer reformulation. The computations on a large set of test instances illustrate that, in general, it is advantageous to generate cuts directly for the robust problem rather than using the reformulation: The amount of gap closed is similar while cut generation is performed faster.

Chapter 2

Preliminaries

2.1 Notation

Tabular lists of notations and abbreviations used are found in the back of this thesis.

We denote with $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ the sets of integer, rational and real numbers, respectively. $\mathbb{Z}_+$, $\mathbb{Q}_+$ and $\mathbb{R}_+$ denote the non-negative integer, rational and real numbers, respectively. $\mathbb{B}$ denotes the set $\{0,1\}$. For a number $n \in \mathbb{Z}_+ \setminus \{0\}$, $[n]$ denotes the set $\{1,\ldots,n\}$. Except for these special sets, other sets are usually denoted by calligraphic upper case letters, for example $\mathcal{S}$.

For a set $\mathcal{S}$ and $m \in \mathbb{Z}_+ \setminus \{0\}$, the set of (column) vectors of length m with entries from $\mathcal{S}$ is denoted by $\mathcal{S}^m$. For an additional $n \in \mathbb{Z}_+ \setminus \{0\}$, the set of row vectors of length n is denoted by $\mathcal{S}^{1,n}$ and the set of matrices with m rows and n columns by $\mathcal{S}^{m,n}$. Column vectors and row vectors are regarded as matrices having only one column or only one row, respectively.

Vectors are usually denoted by lowercase bold letters, like $\boldsymbol{x}$, matrices by upper case bold letters, like $\boldsymbol{A}$. The rows of a matrix $\boldsymbol{A}$ are indexed by subscripts, like $\boldsymbol{a}_i$, columns of $\boldsymbol{A}$ are indexed by superscripts, like $\boldsymbol{a}^j$. Then $\boldsymbol{A} = (\boldsymbol{a}_1;\ldots;\boldsymbol{a}_m) = (\boldsymbol{a}^1,\ldots,\boldsymbol{a}^n)$ with $(\ldots;\ldots)$ denoting vertical and $(\ldots,\ldots)$ horizontal concatenation. The entry at the ith row and jth column is then denoted by a_i^j. Likewise, if $\boldsymbol{x}$ is a column vector, x_i denotes the entry in the ith row, and, if it is a row vector, x^j denotes the entry in the jth column. For a set of row indices $\mathcal{I}$ and a set of column indices $\mathcal{J}$, we denote with $\boldsymbol{A}_{\mathcal{I}}^{\mathcal{J}}$ the submatrix of $\boldsymbol{A}$ built from the entries a_i^j with $i \in \mathcal{I}$ and $j \in \mathcal{J}$; we denote row-wise only and column-wise only submatrices by $\boldsymbol{A}_{\mathcal{I}}$ and $\boldsymbol{A}^{\mathcal{J}}$, respectively, and subvectors by $\boldsymbol{x}_{\mathcal{I}}$ and $\boldsymbol{x}^{\mathcal{J}}$. With $\boldsymbol{A}^\top$ we denote the transposed matrix having entries a_j^i where $\boldsymbol{A}$ has entries a_i^j.

$\boldsymbol{0}_{m,n}$ ($\boldsymbol{0}$ if m and n are clear from context) denotes the all-zero matrix, $\boldsymbol{1}_{m,n}$ ($\boldsymbol{1}$ if m and n are clear from context) denotes the all-one matrix. For a vector $\boldsymbol{d}$, $\operatorname{diag}(\boldsymbol{d})$ denotes the matrix with entries d_i on the diagonal as the only non-zero entries. The matrix $\boldsymbol{E}_n$ ($\boldsymbol{E}$ if n is clear from context) denotes

the identity matrix, $\boldsymbol{E}_n = \operatorname{diag}(\mathbf{1}_{n,1})$. The jth column of $\boldsymbol{E}$ is the jth unit vector and is denoted by $\boldsymbol{e}^j$, the ith row is denoted by $\boldsymbol{e}_i$.

For a vector $\boldsymbol{x}$, the Euclidean norm is denoted by $\|\boldsymbol{x}\|$ and the l_1-norm is denoted by $\|\boldsymbol{x}\|_1$. The convex hull of a set $\mathcal{S}$ is denoted by $\operatorname{conv}(\mathcal{S})$, the closure of the convex hull by $\overline{\operatorname{conv}}(\mathcal{S})$. The convex conic hull is denoted by $\operatorname{ccone}(\mathcal{S})$, its closure by $\overline{\operatorname{ccone}}(\mathcal{S})$. If a set $\mathcal{S}$ is defined in terms of variables $\boldsymbol{x}$ and $\boldsymbol{y}$, then $\operatorname{proj}_{\boldsymbol{x}}(\mathcal{S})$ denotes the projection of $\mathcal{S}$ onto the subspace corresponding to the variable $\boldsymbol{x}$, $\operatorname{proj}_{\boldsymbol{x}}(\mathcal{S}) := \{\boldsymbol{x} : \exists \boldsymbol{y} : (\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{S}\}$.

2.2 Algorithms and Complexity

In this thesis, we will often talk about problems, solution algorithms and their complexity. We will introduce the basic concepts of computational complexity to formalize these notions. Because a complete treatment of this subject would go beyond the scope of this introduction, we refer to the usual books (Garey and Johnson, 1990; Arora and Barak, 2009), in particular to the one by Grötschel, Lovász, and Schrijver (1988), upon which most of the following is based.

2.2.1 Problems and Algorithms

A *problem* is a question that is formulated in dependence on a series of parameters. An *instance* of a problem is an assignment of values to all of the parameters. A *solution* of a given instance is the answer to the problem question using the parameters assigned by the instance. For a problem to be well defined we expect to have an explicit description of the properties that characterize the solutions for a given instance.

To formalize solution algorithms for problems, we need to agree on how problems are represented. Let $\mathbb{B}^*$ the set of finite strings of symbols from $\mathbb{B} = \{0, 1\}$. An *encoding scheme* for a problem is a pair of one-to-one mappings that map instances/solutions to finite strings in $\mathbb{B}^*$. The *(encoding) size* $\langle x \rangle \in \mathbb{Z}_+$ of some object x is the number of digits that is needed to encode x under a specified encoding scheme.[1] A problem is then simply a subset Π of $\mathbb{B}^* \times \mathbb{B}^*$, and $\tau \in \mathbb{B}^*$ is a solution of instance $\sigma \in \mathbb{B}^*$ if and only if $(\sigma, \tau) \in \Pi$. We will assume that for all instances $\sigma \in \mathbb{B}^*$ there is a solution

[1] We always assume the *natural* encoding scheme: An integer $n \in \mathbb{Z}$ is encoded as a binary string using $\langle n \rangle = 1 + \lceil \log_2(|n| + 1) \rceil$ digits. When we group simpler objects into pairs, vectors, sets and more complicated objects, we assume that the size of the compound object is the sum of sizes of its parts (ignoring the digits that would be used to indicate that and how the sting is to be interpreted as a compound object).

$\tau \in \mathbb{B}^*$ such that $(\sigma, \tau) \in \Pi$, possibly by introducing a solution that encodes the answer "no solution".

An algorithm is a set of instructions that can be executed under a deterministic computing model with the possibility of initially reading input and of finally producing output and halt. The *output* of an algorithm M on input σ is denoted by $M(\sigma)$. We say that M *computes* τ from σ if and only if $\tau = M(\sigma)$. An algorithm M is said to *solve* problem Π if and only if $(\sigma, M(\sigma)) \in \Pi$ for all $\sigma \in \mathbb{B}^*$. Formally, the computing model we use is that of a *Turing machine* (see Grötschel et al., 1988, for details). Quite informally it can be thought of as an algorithm running on an ordinary computer, on which basic instructions (read a symbol from $\mathbb{B}$ from the current memory position, write a symbol from $\mathbb{B}$ to the current memory position, move the current memory position forward/backward, change the computers state register) are sequentially executed on a single processor, just with the difference that there is access to unlimited memory. With this excuse we will use *algorithm* as a synonym for Turing machine.

The *time complexity function* $T_M : \mathbb{Z}_+ \to \mathbb{Z}_+ \cup \{\infty\}$ of an algorithm is defined such that $T_M(n) = m$ if and only if m is the maximum number of basic instructions M executes to reach a halt state for inputs of size n. An algorithm M is said to be *polynomial-time* if and only if there is a polynomial p such that $T_M(n) \leq p(n)$ for all $n \in \mathbb{Z}_+$.

2.2.2 Decision Problems and the Classes P and NP

A *decision problem* is a particular kind of problem where the question is of such a type that it can only be answered by "yes" or by "no". To be precise, Π is a decision problem if and only if for all $\sigma \in \mathbb{B}^*$ either $(\sigma, 0) \in \Pi$ or $(\sigma, 1) \in \Pi$, and $(\sigma, \tau) \in \Pi$ implies $\tau \in \mathbb{B}$. The string $\tau = 1$ represents the answer "yes", while $\tau = 0$ means "no". An instance is called a *yes-instance* if its answer is "yes" and a *no-instance* otherwise.

The complexity class P is defined as the class of decision problems Π for which there is a polynomial-time algorithm M such that, for all $\sigma \in \mathbb{B}^*$,

$$(\sigma, 1) \in \Pi \quad \Leftrightarrow \quad M(\sigma) = 1.$$

In other words, M solves Π in polynomial time.

The complexity class NP is defined as the class of decision problems Π for which there is a polynomial-time algorithm M and a polynomial p such that, for all $\sigma \in \mathbb{B}^*$,

$$(\sigma, 1) \in \Pi \quad \Leftrightarrow \quad \exists \tau \in \mathbb{B}^{p(\langle \sigma \rangle)} : M(\sigma, \tau) = 1.$$

For a problem in NP it can be verified in polynomial time that some string τ is a *short certificate* for that the instance is a yes-instance.

The class NP is asymmetrically defined with respect to yes- and no-instances. However, we can as well negate the question of any decision problem to get a new decision problem. For a decision problem Π, the *complementary problem* is $\{(\sigma, 1-\tau) : (\sigma, \tau) \in \Pi\}$. For any class of decision problems C, the problems complementary to the problems in C form the class coC.

2.2.3 Transformations, Oracles and Reductions

It is sometimes desirable to measure the complexity of a problem relative to the complexity of another problem. New knowledge about the complexity of one problem may then shed a new light on the complexity of the other.

A *polynomial transformation* from decision problem Π to decision problem Π' is a polynomial-time algorithm M that takes an instance σ of Π as input and outputs an instance $M(\sigma)$ of Π' such that, for all $\sigma \in \mathbb{B}^*$, the following holds: $M(\sigma)$ is a yes-instance of Π' if and only if σ is a yes-instance of Π. A polynomial transformation from one decision problem Π to another decision problem Π' makes the latter at least as hard as the former: $\Pi' \in \mathsf{P}$ ($\Pi' \in \mathsf{NP}$) implies $\Pi \in \mathsf{P}$ ($\Pi \in \mathsf{NP}$).

For the search for efficient algorithms it makes sense to identify those decision problems that seem to be the hardest within a complexity class C. A problem $\Pi \in \mathsf{C}$ is called C-*complete* if and only if from every problem in C there is a polynomial transformation to Π. A justified question is whether such problems exist at all. With the satisfiability problem SAT, Cook (1971) was the first to identify an NP-complete problem.

SAT: Given a Boolean formula[2] φ in CNF with n variables, decide whether

$$\exists \boldsymbol{x} \in \mathbb{B}^n : \varphi(\boldsymbol{x}) = 1.$$

[2] For an introduction to Boolean formulae and logic we refer to the literature (Kleine Büning and Lettman, 1999, for instance). In short, a *Boolean formula* is recursively defined as: *variables* α_j are Boolean formulae, and if φ and φ_j, $j \in \mathcal{M}$ with finite $\mathcal{M}$, are Boolean formulae, then so are the *negation* $\neg\varphi$, the *conjunction* $\bigwedge_{j\in\mathcal{M}} \varphi_j$ and the *disjunction* $\bigvee_{j\in\mathcal{M}} \varphi_j$. A *literal* is a Boolean formula that is either a variable or the negation of a variable. A Boolean formula in *CNF* is a conjunction of disjunctions of literals, a Boolean formula in *DNF* is a disjunction of conjunctions of literals. We can assign *truth values* $T(\alpha_j) \in \mathbb{B}$ to the variables α_j, $j \in [n]$, and then evaluate the truth value $T(\varphi) \in \mathbb{B}$ of the formula φ. If $\boldsymbol{x} \in \mathbb{B}^n$ is a vector of truth values and $T(\alpha_j) = x_j$ for all $j \in [n]$, we define $\varphi(\boldsymbol{x}) := T(\varphi)$. If $\varphi(\boldsymbol{x}) = 1$, then T is called a *satisfying truth assignment*.

There is an extensive list of problems now known to be NP-complete (Garey and Johnson, 1990). The significance of the existence of NP-complete problems is that, if any of them turns out to be in P, then P = NP.

Often, problems are solved by calling algorithms for other problems as a subroutine. An *algorithm with an oracle for problem* Π is an algorithm augmented by the following feature: The algorithm can, as often as it needs to, call the oracle for an instance σ of Π, and the oracle returns the solution τ for this instance. It is assumed that this miraculously happens within one time step and that the size of the solution is bounded by a polynomial in the size of the instance. Because the call to the oracle counts as one time step, an algorithm with an oracle can be polynomial-time even if the oracle solves a problem not known be solved by a polynomial-time algorithm.

A *Turing reduction* from Π to Π' is a polynomial-time algorithm with an oracle for Π' that solves Π. We say that Π is *Turing reducible* to Π' if and only if there is a Turing reduction from Π to Π'. Turing reductions can be applied to general problems, not just decision problems. For any complexity class C, a problem Π is called C-*hard* if and only if there is a Turing reduction from some C-complete problem to Π. A problem Π is called C-*easy* if and only if there is a Turing reduction from Π to some problem in C. A problem is called C-*equivalent* if and only if it is C-easy and C-hard.

2.2.4 Beyond NP and the Polynomial Hierarchy

With our definition of NP-hardness we also captured grossly all decision problems that we believe to be at least as hard as those in NP. In fact, there are different levels of hardness above NP.

With P^{NP} we denote the decision problems that are Turing reducible to a problem in NP. These are exactly the NP-easy decision problems. Clearly $\mathsf{NP} \subseteq \mathsf{P}^{\mathsf{NP}}$. There are P^{NP}-complete problems: Wagner (1987) proved that $\mathsf{MaxSAT}_{\mathsf{odd}}$ is one.

> $\mathsf{MaxSAT}_{\mathsf{odd}}$: Given a Boolean formula φ in CNF with n variables, decide whether the optimization problem
>
> $$\max\{\sum_{j\in[n]} 2^{j-1}x_j : \varphi(\boldsymbol{x}) = 1,\ \boldsymbol{x} \in \mathbb{B}^n\}$$
>
> is feasible and the optimal objective is odd.

An infinite hierarchy of problems above NP was introduced by Meyer