# idol: A C++ Framework for Optimization

## Reference Manual

For idol v1.0.0-beta.

Henri Lefebvre

# Contents

# Part 1

# Mixed-integer Optimization

CHAPTER 1

# Modeling a MIP with idol

## 1. Introduction

In many decision-making applications—ranging from logistics and finance to energy systems and scheduling—problems can be naturally modeled as mixed-integer optimization problems (MIPs). These problems combine continuous and integer variables to represent decisions under various logical, structural, or operational constraints.

In idol, we adopt a general and flexible framework for expressing such problems. A MIP is assumed to be of the following form:

$$\min_{x} \quad c^\top x + x^\top D x + c_0 \tag{1a}$$

$$\text{s.t.} \quad a_{i\cdot}^\top x + x^\top Q^i x \le b_i, \quad \text{for all } i = 1, \dots, m, \tag{1b}$$

$$\ell_j \le x_j \le u_j, \quad \text{for all } j = 1, \dots, n, \tag{1c}$$

$$x_j \in \mathbb{Z}, \quad \text{for all } j \in J \subseteq \{1, \dots, n\}. \tag{1d}$$

Here, $x$ is the decision variable vector, and the input data are as follows: Vector $c \in \mathbb{Q}^n$, matrix $D \in \mathbb{Q}^{n\times n}$ and the constant $c_0 \in \mathbb{Q}$ define the linear, the quadratic and the constant parts of the objective function, respectively; For each constraint with index $i \in \{1, \dots, m\}$, vector $a_{i\cdot}$, matrix $Q^i \in \mathbb{Q}^{n\times n}$ and constant $b_i$ encode the linear part, the quadratic part, and the right-hand side of the constraint respectively; Vectors $\ell \in \mathbb{Q}^n \cup \{-\infty\}$ and $u \in \mathbb{Q}^n \cup \{\infty\}$ are used to define lower and upper bounds on each variables; Finally, the set $J \subseteq \{1, \dots, n\}$ specifies which variables are required to be integer.

As is customary, variables are classified depending on their type—which can be continuous, integer or binary—and bounds. This is presented in Table 1. As to constraints, they are said to be linear when $Q^i = 0$, and quadratic otherwise. Likewise, the objective function is quadratic when $D \neq 0$.

A particularly important subclass of MIPs arises when both the constraints and the objective function are linear (i.e., $Q^i = 0$ for all $i$ and $D = 0$). In this case, the problem is known as a mixed-integer linear problem (MILP).

TABLE 1. Terminology for variables in a MIP.

| A variable $x_j$ is said ... | if it satisfies ... |
|---|---|
| integer | $j \in J$ |
| binary | $j \in J$ and $0 \le \ell \le u \le 1$ |
| continuous | $j \notin J$ |
| free | $l = -\infty$ and $u = \infty$ |
| non-negative | $\ell \ge 0$ |
| non-positive | $u \le 0$ |
| bounded | $-\infty < \ell \le u < \infty$ |
| fixed | $\ell = u$ |

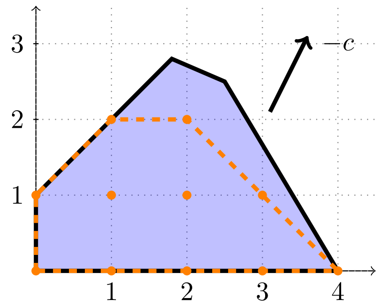


FIGURE 1. Feasible region (shaded in blue) for Problem (2), with integer-feasible points highlighted and objective direction shown.

For many practical purposes, it is helpful to consider the continuous relaxation of a MIP. This is obtained by removing the integrality constraints (1d), allowing all variables to take real values. This relaxation is easier to solve and often provides useful bounds or insights about the original problem.

**Disclaimer.** You do not need to read every section in detail before solving your first problem. Most users can start with the example in Section 2, which demonstrates how to define a basic MILP using idol. Later sections dive deeper into the modeling framework, including variables, constraints, and the environment that manages them. Note that solving a model (i.e., computing an optimal point) is not covered in this chapter. This is the focus of the next chapter, where you will learn about how to use an Optimizer and an OptimizerFactory.

## 2. Example: A mixed-integer linear problem

To illustrate the modeling capabilities of idol, we begin with a small example which is a mixed-integer linear program (MILP). The mathematical model reads:

$$\min_{x} \quad -x - 2y \tag{2a}$$
$$\text{s.t.} \; -x + y \leq 1, \tag{2b}$$
$$2x + 3y \leq 12, \tag{2c}$$
$$3x + 2y \leq 12, \tag{2d}$$
$$x, y \in \mathbb{Z}_{\geq 0}. \tag{2e}$$

This is a minimization problem which involves two integer variables, $x$ and $y$, both constrained to be non-negative. The feasible region is defined by three linear inequalities. Figure 1 shows this region (shaded in blue), the integer-feasible points, and the direction of the objective function (represented by the vector $-c$). The continuous relaxation of this problem—where $x$ and $y$ are allowed to take real values—has a unique solution at $(x^*, y^*) = (2.4, 2.4)$, with an objective value of $-7.2$. The original integer-constrained problem also admits a unique solution at $(x^*, y^*) = (2, 2)$, with objective value $-5$.

Modeling Problem (2) in idol is straightforward. In particular, if you are familiar with other optimization packages like JuMP in Julia or the Gurobi C++ API, the following code snippet should be easy to understand.

```cpp
#include <iostream>
#include "idol/modeling.h"

using namespace idol;

```

```
6 int main(int t_argc, const char** t_argv) {
7
8     Env env;
9
10    // Create a new model.
11    Model model(env);
12
13    // Create decision variables x and y.
14    const auto x = model.add_var(0, Inf, Integer, -1, "x");
15    const auto y = model.add_var(0, Inf, Integer, -2, "y");
16
17    // Create constraints.
18    const auto c1 = model.add_ctr(-x + y <= 1);
19    const auto c2 = model.add_ctr(2 * x + 3 * y <= 12);
20    const auto c3 = model.add_ctr(3 * x + 2 * y <= 12);
21
22    return 0;
23 }
```

Let's walk through this code. In Line 8, we create a new optimization environment that will store all our optimization objects such as variables or constraints. Destroying an environment automatically destroys all objects which were created with this environment. Then, in Line 11, we create an optimization model. By default, all models are for minimization problems. Decision variables are created in Lines 14 and 15. There, we set the lower bound to 0 and an infinite upper bound using the defined constant idol::Inf. Both variables are defined as Integer. Note that other types are possible, e.g., Continuous and Binary. The objective coefficients are also set in these lines by the fourth argument. The last argument corresponds to the internal name of that variable and is mainly useful for debugging. Finally, Lines 18–20 add constraints to the model to define the feasible region of the problem.

Note that, as such, we are only modeling Problem (2) here but we are not performing any optimization task so far. This is the subject of the next chapter. For the sake of completeness, however, here is how one uses the commercial solver Gurobi to compute a solution of this problem.

```
    model.use(Gurobi());
    model.optimize();

    std::cout << "Status = " << model.get_status() << std::endl;
    std::cout << "x = " << model.get_var_primal(x) << std::endl;
    std::cout << "y = " << model.get_var_primal(y) << std::endl;
```

This prints the solution status (here, Optimal) and the values of the decision variables in the solution (here, $(x^*, y^*) = (2, 2)$).

## 3. The environment

Any optimization object—such as variables, constraints and models—are managed through a central entity called an "optimization environment". This environment, represented by the Env class, acts as a container and controller for all optimization-related objects created within its scope.

The environment has two key responsabilities:

(1) **Lifecycle management.** When an environment is destroyed, all objects created within it are automatically deleted. This eliminates the need for manual memory management. Also, once an object is no longer referenced, it is safely cleaned up by the environment, i.e., you do not need to manually delete objects.

(2) **Version tracking.** During the execution of an optimization program, objects like variables and constraints may appear in different models with model-specific changes. These different versions of a single object are all stored and managed in the environment.

Typically, a single environment should suffice for most applications. While idol technically allows the creation of multiple environments, this is strongly discouraged. Objects created in one environment must not be mixed with those from another. For example, attempting to add a variable from one environment to a model belonging to a different environment will lead to undefined behavior, often resulting in segmentation faults or program crashes.

Creating an environment is straightforward:

```
Env env; // Creates a new optimization environment.
```

Once initialized, you can begin creating models, variables, and constraints using this environment. All such objects will be associated with env and managed accordingly throughout their lifetime.

## 4. Models

Mathematical optimization problems are modeled using the Model class. A model is a set of variables and constraints with an objective function. It can be created by calling the constructor of the Model class by passing an environment as the first argument.

```
Env env;
Model model(env); // Creates an empty model.
```

Here, we first create a new optimization environment, then create an optimization model. Note that the newly created model does not contain any variable nor constraints. By default, all models are for minimization problems. Unfortunately, idol offers limited support for maximization problem. However, it is well knwon that this is not a real restriction since $\max f(x) = -\min -f(x)$.

Another way to create a model is by importing it from an .mps or an .lp file. To do this, you will need to rely on an external solver. In what follows, we use GLPK, which is an open-source solver which can be easily installed on your computer.

```
Env env;
auto model = GLPK::read_from_file("/path/to/some/file.mps");
```

The choice to rely on external solver is justified by the fact that, first of all, idol will most of the time be used in combination with such a solver to effectively solve optimization problems. Second, it is also safer to rely on existing codes with years of experience to import your model without mistake or ambiguity.

Now that we have a model imported, we can safely iterate over its variables and constraints. This can be done as follows.

```
for (const auto& var : model.vars()) {
    std::cout << var.name() << std::endl;
}
```

Here, we use the Model::vars() method to get access to the variables of the model and write down their names. Note that you can also use the operator«(std::ostream&, const Model&) function to print the model to the console. This can be useful for debugging.

Once we have iterated over the variables, we may want to iterate over constraints as well. To do so, we can use: the Model::ctrs() method for linear constraints, the Model::qctrs() method for quadratic constraints and the Model::sosctrs() method for SOS-type constraints. The next code snippet shows how to get the number of variables and constraints.

```
std::cout << "Number of variables: "
          << model.vars().size() << std::endl;

std::cout << "Number of linear constraints: "
          << model.ctrs().size() << std::endl;

std::cout << "Number of quadratic constraints: "
          << model.qctrs().size() << std::endl;

std::cout << "Number of sos-type constraints: "
          << model.sosctrs().size() << std::endl;
```

To get model-specific information about a variable, a constraint or the objective function, we can use methods like Model::get_X_Y(const X&) where X is an object name—like var, ctr, qctr, sosctr or obj—and Y is the data name you which to gain access—like lb, type or column. For instance, the following code counts the number of binary variables in the model.

```
unsigned int n_binary_vars = 0;

// Iterate over all variables in the model.
for (const auto& var : model.vars()) {

    // Get the variable type in this model.
    const auto type = model.get_var_type(var);

    // Check type is binary.
    if (type == Binary) {
        ++n_binary_vars;
    }

}
```

Complete details on what information you can retrieve through a model will be detailed in the following sections.

In most practical cases, you will want to avoid copying a model but, rather, pass on a reference to some auxiliary function. For that reason, the copy constructor of the Model class is declared as private. If copying a model is really what you intend to do, you should use the Model::copy() method and the move constructor. Here is an example.

```
const auto model = Gurobi::read_from_file("problem.lp");
auto model2 = model.copy();
```

Here, model2 is now an independent copy of the original model and can be modified without altering its source model. Similarly, if you want to write a function that returns a model, you will have to be explicit about it to avoid unncessary copies. See the following code.

```
Model read_model_from_file() {

    // Read the model from the file.
    auto model = Gurobi::read_from_file("problem.lp");

    // Use std::move to avoid unnecessary copies.
    // If a copy is intended, use Model::copy().
    return std::move(model);
}
```

Moving the model instead of copying it avoids the overhead of duplicating large optimization problems.

## 5. Variables

Variables are the decision-making elements of an optimization problem. These are the quantities that we aim to determine in order to optimize an objective function, subject to a set of constraints. In idol, they are represented by the Var class.

**5.1. Creating variables.** Creating variables can be done in mainly two ways. The first one is through the Var constructor and the Model::add() method, while the second uses the Model::add_var(...) methods. We start with the fist method which uses the Var constructor. This method is less direct, but more informative on how optimization objects are managed in idol. We focus on the following constructor:

Var(Env&, double, double, VarType, double, std::string).

This constructor takes six arguments. The first is the optimization environment which will store the variable's versions. The two subsequent are the lower and upper bound—possibly infinite using idol::Inf. Then, the type of the variable is expected—such as idol::Continuous, idol::Integer or idol::Binary. The linear coefficient of the variable in the objective function is the fifth argument. Finally, the last argument is the given name of the variable. For instance, the following code creates a new variable in the environment.

```
Var x(env, 0, Inf, Continuous, 2, "x");
```

This variable is a continuous non-negative variable with an objective coefficient of 2. It is called "x". One important thing is that this variable does not belong to any model yet. Instead, what we have created is called the "default version" of the variable. This means that, by default, if this variable is added to a model, it will have the corresponding attributes in that model. For instance, here is a code that creates and add this variable to a model.

```
// Create a variable in the environment.
Var x(env, 0, Inf, Continuous, 2, "x");

// Add the variable to a model
model.add(x);
```

By default, the variable "x" is added to the model as a continuous non-negative variable with an objective coefficient of 2. Note that other constructors are also available in the Var class. For instance, it is also possible to provide a column associated to the variable so that it is automatically added to the LP matrix. Columns are built using the LinExpr<Ctr> class and can be built in a very natural way. For more details, please refer to Section 6 on expressions in idol. We simply give one example.

```
// This function is assumed to return a vector of constraints.
const std::vector<Ctr> ctrs = get_vector_of_ctrs();

// Create the column associated to x.
LinExpr<Ctr> column = -1 * c[0] + 2 * c[1] + 3 * c[2];

// Create a variable in the environment.
Var x(env, 0, Inf, Integer, -1, std::move(column), "x");

// Add the variable to a model.
model.add(x);
```

Finally, note that it is possible to avoid adding the default version to a model by overriding it as follows.

```
// Add the variable to a model, overriding the default version.
model.add(x, TempVar(0, Inf, Continuous, 2, LinExpr<Var>()));
```

Here, we notice the use of the TempVar class. This class is a lightweight class used to represent a variable that has yet not been created inside an environment. As such, it contains all attributes of the variable to be created but it cannot be used other than for storing these attributes and create an actual variable.

The second approach for creating variables is more straightforward. However, it internally is exactly the same as what we have seen so far. This can be reached using the Model::add_var methods from the Model class. The following code snippet should be easy to understand.

```
const auto x = model.add_var(0, Inf, Continuous, 2, "x");
```

Note that we do not need to pass the environment since the environment of the model is automatically used. Also, two operations are performed in a single call here: first, a default version is created for the variable, then the variable is added to the model. Similarly, it is also possible to add a variable with a specific column in the LP matrix.

Sometimes, you will find it more convenient to create several variables at once. This can be done by calling the Var::make_vector function, or the Model::add_vars method. These functions require one extra parameter specifying the dimension of the new variable. For instance, here is how to create a set of variables with a $2 \times 3$ index.

```
// Create a 2x3 "vector" of variables.
const auto x = Var::make_vector(env, Dim<2>(2, 3), 0, Inf, Continuous, "x");

// Add all variables
model.add_vector<Var, 2>(x);

// Print the first variable's name.
std::cout << "x_0_0 = " << x[0][0].name() << std::endl;
```

Notice that we used the Dim class to specify the dimensions. The Dim class is a template class that takes an integer as parameter. This integer specifies the number of indices for the new variable. In this case, we use 2 to specify that we want to create a two-dimensional index. Then, we give the size of each dimension by passing the appropriate arguments to the constructor of the Dim class, i.e., 2 and 3.

Naturally, it is also possible to achieve this goal through methods of the Model class. The following snippet gives an example.

```
const auto x = model.add_vars(Dim<2>(2,3), 0, Inf, Continuous, "x");
```

**5.2. Removing variables.** Once a variable has been added to a model, it can also be removed from it. We use the Model::remove(const Var&) method for this. Calling this method will remove the variable from the model and update all linear and quadratic constraints where this variable appeared. Trying to remove a variable which does not belong to a model will result in an exception being thrown. However, it is possible to check whether a model has a given variable using the Model::has(conv Var&) method. This method returns true if and only if the variable is part of the model. Also, note that it is not possible to remove a variable which is involved in an SOS-type constraint. This is not limiting since SOS-type constraints can be removed and added again.

**5.3. Accessing variables.** Variables have two immutable attributes: a name, which is the given name at creation time of the variable and an id, which is unique in the environment. Other attributes are tied to a specific model and can be accessed through the model's methods Model::get_var_Y where Y is the name of that attribute. Next is list of methods which can be used to retrieve information about variables in a model.

**double Model::get_var_lb(const Var&):**
Returns the lower bound of the variable given as parameter.
May return any value between -idol::Inf and idol::Inf.

**double Model::get_var_ub(const Var&):**
Returns the upper bound of the variable given as parameter.
May return any value between -idol::Inf and idol::Inf.

**double Model::get_var_obj(const Var&):**
Returns the objective coefficient in the linear part of the objective function.

**VarType Model::get_var_type(const Var&):**
Returns the type of the variable which can be Continuous, Integer or Binary.

**LinExpr<Ctr> Model::get_var_column(const Var&):**
Returns the associated column in the LP matrix.

**unsigned int Model::get_var_index(const Var&):**
Returns the index of the variable.
Note that this index may change if variables are removed.

We now give an example which prints out all free variables.

```cpp
for (const auto& var : model.vars()) {

    const double lb = model.get_var_lb(var);
    const double ub = model.get_var_ub(var);

    if (is_neg_inf(lb) && is_pos_inf(ub)) {
        std::cout << var.name() << " is free." << std::endl;
    }

}
```

One final note regarding indices. Though they may change over time, e.g., if variables are removed from a model, it can still be used to access variables by using the Model::get_var_by_index method. The following code snippet shows an alternative way to iterate over variables in a model.

```cpp
for (unsigned int i = 0, n = model.vars().size(); i < n; ++i) {

    // Get the variable by index
    const auto& var = model.get_var_by_index(i);

    // Print out its name
    std::cout << var.name() << std::endl;

}
```

**5.4. Modifying variables.** Some of the attributes of a variable may be directly changed through the model's methods Model::set_var_Y. Here again, Y is the name of the attribute you wish to modify. Here is a list of methods to be used for modifying attributes of a variable in a model.

**void Model::set_var_lb(const Var&, double):**
Sets the lower bound of a variable.
The new lower bound can be -idol::Inf, idol::Inf or any double in between.

**void Model::set_var_ub(const Var&, double):**
Sets the lower bound of a variable.
The new lower bound can be -idol::Inf, idol::Inf or any double in between.

**void Model::set_var_obj(const Var&, double):**
Sets the linear coefficient in the objective function.

**void Model::set_var_type(const Var&, VarType):**
Sets the type of a variable.
Changing the type of variable does not affect its bounds.

**void Model::set_var_column(const Var&, const LinExpr<Ctr>&):**
Sets the column of a variable in the LP matrix.

We end with an example which copies a model and creates its continuous relaxation.

```
// Copy the model.
auto continuous_relaxation = model.copy();

// Build the continuous relaxation.
for (const auto& var : model.vars()) {
    continuous_relaxation.set_var_type(var, Continuous);
}
```

## 6. Expressions

## 7. Constraints

### 7.1. Linear constraints.

### 7.2. Quadratic constraints.

### 7.3. SOS1 and SOS2 constraints.

## 8. The objective function

CHAPTER 2

# Solving problems with Optimizers

CHAPTER 3

# Callbacks

CHAPTER 4

# Writing a custom Branch-and-Bound algorithm

CHAPTER 5

# Column Generation and Branch-and-Price

CHAPTER 6

# Penalty alternating direction method

# Part 2

# Bilevel Optimization

CHAPTER 7

# Modeling a bilevel problem

CHAPTER 8

# Problems with continuous follower

$$
\begin{aligned}
\min_{x,y} \quad & c^\top x + d^\top y && \text{(3a)}\\
\text{s.t.} \quad & Ax + By \geq a, && \text{(3b)}\\
& y \in S(x). && \text{(3c)}
\end{aligned}
$$

## 1. The strong-duality single-level reformulation

## 2. The KKT single-level reformulation

## 3. Linearization techniques for the KKT single-level reformulation

$$
\begin{aligned}
\min_{y} \quad & f^\top y && && \text{(4a)}\\
\text{s.t.} \quad & C^{=}x + D^{=}y = b^{=}, && (\lambda^{=} \in \mathbb{R}^{m_{=}}) && \text{(4b)}\\
& C^{\leq}x + D^{\leq}y \leq b^{\leq}, && (\lambda^{\leq} \in \mathbb{R}^{m_{\leq}}_{\leq 0}) && \text{(4c)}\\
& C^{\geq}x + D^{\geq}y \geq b^{\geq}, && (\lambda^{\geq} \in \mathbb{R}^{m_{\geq}}_{\geq 0}) && \text{(4d)}\\
& y \leq y^{\leq}, && (\pi^{\leq} \in \mathbb{R}^{n}_{\leq 0}) && \text{(4e)}\\
& y \geq y^{\geq} && (\pi^{\geq} \in \mathbb{R}^{n}_{\geq 0}). && \text{(4f)}
\end{aligned}
$$

$$
\begin{aligned}
\max_{\lambda^{=},\lambda^{\geq},\lambda^{\leq},\pi^{\leq},\pi^{\geq}} \quad & (b^{=} - C^{=}x)^\top \lambda^{=} + (b^{\leq} - C^{\leq}x)^\top \lambda^{\leq} + (b^{\geq} - C^{\geq}x)^\top \lambda^{\geq} && \text{(5a)}\\
& \quad + \sum_{j:y_j^{\leq} < \infty} (y^{\leq})^\top \pi^{\leq} + \sum_{j:y_j^{\geq} > -\infty} (y^{\geq})^\top \pi^{\geq} && \text{(5b)}\\
\text{s.t.} \quad & (D^{=})^\top \lambda^{=} + (D^{\leq})^\top \lambda^{\leq} + (D^{\geq})^\top \lambda^{\geq} + \pi^{\leq} + \pi^{\geq} = d, && \text{(5c)}\\
& \lambda^{\leq} \leq 0, \lambda^{\geq} \geq 0, \pi^{\leq} \leq 0, \pi^{\geq} \geq 0. && \text{(5d)}
\end{aligned}
$$

The KKT system reads

$$\text{Primal feasibility} \quad \begin{cases} C^{=}x + D^{=}y = b^{=}, \\ C^{\leq}x + D^{\leq}y \leq b^{\leq}, \\ C^{\geq}x + D^{\geq}y \geq b^{\geq}, \\ y \leq y^{\leq}, \\ y \geq y^{\geq}, \end{cases}$$

$$\text{Dual feasibility} \quad \begin{cases} \lambda^{\leq} \leq 0, \\ \lambda^{\geq} \geq 0, \\ \pi^{\leq} \leq 0, \\ \pi^{\geq} \geq 0, \end{cases}$$

$$\text{Stationarity} \quad \left\{ (D^{=})^{\top}\lambda^{=} + (D^{\leq})^{\top}\lambda^{\leq} + (D^{\geq})^{\top}\lambda^{\geq} + \pi^{\leq} + \pi^{\geq} = d, \right.$$

$$\text{Complementarity} \quad \begin{cases} (C^{\leq}x + D^{\leq} - b^{\leq})^{\top}\lambda^{\leq} = 0, \\ (C^{\geq}x + D^{\geq} - b^{\geq})^{\top}\lambda^{\geq} = 0, \\ (y - y^{\leq})^{\top}\pi^{\leq} = 0, \\ (y - y^{\geq})^{\top}\pi^{\geq} = 0. \end{cases}$$

**3.1. Using SOS1 constraints.**

$$\begin{aligned} &(C^{\leq}x + D^{\leq} - b^{\leq}) = s^{\leq}, \\ &(C^{\geq}x + D^{\geq} - b^{\geq}) = s^{\geq}, \\ &(y - y^{\leq}) = r^{\leq}, \\ &(y - y^{\geq}) = r^{\geq}, \\ &\mathrm{SOS1}(s_i^{\leq}, \lambda_i^{\leq}), \quad \text{for all } i = 1, \dots, m_{\leq}, \\ &\mathrm{SOS1}(s_i^{\geq}, \lambda_i^{\geq}), \quad \text{for all } i = 1, \dots, m_{\geq}, \\ &\mathrm{SOS1}(r_i^{\leq}, \pi_i^{\leq}), \quad \text{for all } i = 1, \dots, n, \\ &\mathrm{SOS1}(r_i^{\geq}, \pi_i^{\geq}), \quad \text{for all } i = 1, \dots, n. \end{aligned}$$

**3.2. Using the big-M approach.**

$$\begin{aligned} &M_i^{\leq}u_i^{\leq} \leq \lambda^{\leq} \leq 0, \quad N_i^{\leq}(1 - u_i^{\leq}) \leq C^{\leq}x + D^{\leq}y - b^{\leq} \leq 0, \quad \text{for all } i = 1, \dots, m_{\leq}, \\ &M_i^{\geq}u_i^{\geq} \geq \lambda^{\geq} \geq 0, \quad N_i^{\geq}(1 - u_i^{\geq}) \geq C^{\geq}x + D^{\geq}y - b^{\geq} \geq 0, \quad \text{for all } i = 1, \dots, m_{\geq}, \\ &O_j^{\leq}v_j^{\leq} \leq \pi^{\leq} \leq 0, \quad P_i^{\leq}(1 - v_j^{\leq}) \leq y - y^{\leq} \leq 0, \quad \text{for all } j = 1, \dots, n, \\ &O_j^{\geq}v_j^{\geq} \geq \pi^{\geq} \geq 0, \quad P_i^{\leq}(1 - v_j^{\leq}) \geq y - y^{\geq} \geq 0, \quad \text{for all } j = 1, \dots, n, \\ &u^{\leq} \in \{0,1\}^{m_{\leq}}, \quad u^{\geq} \in \{0,1\}^{m_{\geq}}, \quad v^{\leq} \in \{0,1\}^{n}, \quad v^{\geq} \in \{0,1\}^{n}. \end{aligned}$$

## 4. Penalty alternating direction methods

| CtrType | | |
|---|---|---|
| LessOrEqual | $M_i^{\leq} \leftarrow$ get_ctr_dual_lb(c) | $N_i^{\leq} \leftarrow$ get_ctr_slack_lb(c) |
| GreaterOrEqual | $M_i^{\geq} \leftarrow$ get_ctr_dual_ub(c) | $N_i^{\geq} \leftarrow$ get_ctr_slack_ub(c) |

| Var | |
|---|---|
| $O_j^{\leq} \leftarrow$ get_var_ub_dual_lb(y) | $P_j^{\leq} \leftarrow y^{\geq} - y^{\leq}$ |
| $O_j^{\geq} \leftarrow$ get_var_lb_dual_ub(y) | $P_j^{\geq} \leftarrow y^{\leq} - y^{\geq}$ |

TABLE 1. Function calls made to the BoundProvider to linearize a KKT single-level reformulation with the big-M approach.

CHAPTER 9

# Problems with mixed-integer follower

CHAPTER 10

# Pessimistic bilevel optimization

# Part 3

# Robust Optimization

CHAPTER 11

# Modeling a robust problem

## 1. Single-stage problems

## 2. Two-stage problems

$$\min_{x \in X} \; c^\top x + \max_{u \in U} \; \min_{y \in Y(x,u)} \; d^\top y$$

$$X := \left\{ x \in \tilde{X} \colon Ax \geq a \right\}$$

$$Y(x,u) := \left\{ y \in \tilde{Y} \colon Cx + Dy + Eu \geq b \right\}$$

$$U := \left\{ u \in \tilde{U} \colon Fu \leq g \right\}$$

$$\begin{aligned}
\min_{x, x_0} \quad & c^\top x + x_0 \\
\text{s.t.} \quad & x \in X, \\
& \forall x \in X, \; \exists y \in Y(x,u), \; x_0 \geq d^\top y.
\end{aligned}$$

## 3. Bilevel problems with wait-and-see follower

$S(x)$ the set of optimal solutions to the follower problem

$$\min_{y \in Y(x,u)} \; f^\top y.$$

$$\min_{x \in X} \; c^\top x + \max_{u \in U} \; \min_{y} \; \left\{ d^\top y \colon y \in S(x,u), \; Gx + Hy + Ju \geq g \right\}.$$

$$\begin{aligned}
\min_{x, x_0} \quad & c^\top x + x_0 \\
\text{s.t.} \quad & x \in X, \\
& \forall x \in X, \; \exists y \in S(x,u), \; x_0 \geq d^\top y, \; Gx + Hy + Ju \geq h.
\end{aligned}$$

CHAPTER 12

# Deterministic reformulations

CHAPTER 13

# Affine decision rules

CHAPTER 14

# Column and constraint generation

## 1. Introduction

## 2. Two-stage robust problems

## 3. Example: The robust uncapacitated facility location problem with facility disruption

$$
\begin{aligned}
\min_{x,y} \quad & \sum_{i \in V_1} f_i x_i + \sum_{i \in V_1} \sum_{j \in V_2} c_{ij} y_{ij} && \text{(8a)} \\
\text{s.t.} \quad & \sum_{i \in V_1} y_{ij} = 1, \quad \text{for all } j \in V_2, && \text{(8b)} \\
& y_{ij} \leq x_i, \quad \text{for all } i \in V_1, && \text{(8c)} \\
& y_{ij} \geq 0, \quad \text{for all } i \in V_1, j \in V_2, && \text{(8d)} \\
& x_i \in \{0,1\}, \quad \text{for all } i \in V_1. && \text{(8e)}
\end{aligned}
$$

$$
Y(x,u) := \left\{ y \in \mathbb{R}^{|V_1| \times |V_2|} \colon \text{(8b)–(8d) and } y_{ij} \leq 1 - \xi_i, \quad \text{for all } i \in V_1 \right\}.
$$

$$
U := \left\{ u \in \{0,1\}^{|V_1|} \colon \sum_{i \in V_1} u_i \leq \Gamma \right\}.
$$

$$
\min_{x \in \{0,1\}^{|V_1|}} \left\{ \sum_{i \in V_1} f_i x_i + \max_{u \in U} \min_{y \in Y(x,u)} \sum_{i \in V_1} \sum_{j \in V_2} c_{ij} y_{ij} \right\}
$$

### 3.1. Solving the separation problem.

3.1.1. *Feasibility separation.* The KKT conditions read

$$
\text{primal feasibility} = \begin{cases} \sum_{i \in V_1} y_{ij} = 1, & \text{for all } j \in V_2, \\ y_{ij} \leq x_i(1 - \xi_i), & \text{for all } i \in V_1, \\ y_{ij} \geq 0, & \text{for all } i \in V_1, j \in V_2, \end{cases}
$$

$$
\text{dual feasibility} = \begin{cases} \beta_{ij} \leq 0, & \text{for all } i \in V_1, \text{for all } j \in V_2, \\ \gamma_{ij} \geq 0 & \text{for all } i \in V_1, \text{for all } j \in V_2, \end{cases}
$$

$$
\text{stationarity} = \begin{cases} \alpha_j + \beta_{ij} + \gamma_{ij} = c_{ij}, & \text{for all } i \in V_1, \text{for all } j \in V_2, \end{cases}
$$

$$
\text{complementarity} = \begin{cases} \beta_{ij}(y_{ij} - x_i + x_i \xi_i) = 0, & \text{for all } i \in V_1, \text{for all } j \in V_2, \\ \gamma_{ij} y_{ij} = 0, & \text{for all } i \in V_1, \text{for all } j \in V_2. \end{cases}
$$

Strong duality implies

$$
\sum_{j \in V_2} \alpha_j + \sum_{i \in V_1} \sum_{j \in V_2} x_i(1 - x_i)\beta_{ij} = \sum_{i \in V_1} \sum_{j \in V_2} c_{ij} y_{ij}.
$$

PROPOSITION 1. *Bounds*

PROOF. We first derive bonds on $\alpha$. First, consider any $(i, j) \in V_1 \times V_2$ with $x_i(1 - u_i) = 1$ and assume that $y_{ij} < 1$. Then, by the complementarity constraints, it holds that $\beta_{ij} = 0$. Hence, the stationarity conditions read

$$\alpha_j + \underbrace{\beta_{ij}}_{=0} + \underbrace{\gamma_{ij}}_{\geq 0} = c_{ij} \implies \alpha_j \leq c_{ij}.$$

Hence, the bound $\alpha_j \leq \min\{c_{ij} : i \in V_1\}$ is valid. If $y_{ij} = 1$, then complementarity conditions imply $\gamma_{ij} = 0$. Hence, stationarity conditions read

$$\alpha_j + \beta_{ij} + \underbrace{\gamma_{ij}}_{=0} = c_{ij} \implies \alpha_j + \beta_{ij} \leq c_{ij}.$$

Note that $\beta_{ij}$ only appears in this constraint and not in the objective function. Hence, fixing it to 0 does not change the optimal objective value. Next, we show bounds on $\beta_{ij}$. To this end, note that if $y_{ij} > 0$, then $\gamma_{ij} = 0$ by the complementarity constraints. Otherwise, $y_{ij} = 0$. Then it only appears in the objective function if it not interdiction, but if it not interdicted $y_{ij} = 0$ leads to $\beta_{ij} = 0$. Hence,

$$\beta_{ij} \geq c_{ij} - \min\{c_{ij} : i \in V_1\}.$$

$\square$

## 4. Example: The capacitated facility location problem with facility disruption

### 4.1. Problem statement.

$$\min_{x,y} \quad \sum_{i\in V_1} f_i x_i + \sum_{i\in V_1}\sum_{j\in V_2} c_{ij} y_{ij} \tag{9a}$$

$$\text{s.t.} \quad \sum_{i\in V_1} y_{ij} = 1, \quad \text{for all } j \in V_2, \tag{9b}$$

$$\sum_{j\in V_2} d_j y_{ij} \leq q_i x_i, \quad \text{for all } i \in V_1, \tag{9c}$$

$$y_{ij} \geq 0, \quad \text{for all } i \in V_1, j \in V_2, \tag{9d}$$

$$x_i \in \{0,1\}, \quad \text{for all } i \in V_1. \tag{9e}$$

$$Y(x,u) := \left\{ y \in \mathbb{R}^{|V_1|\times|V_2|} \colon \text{(9b)–(9d) and } y_{ij} \leq 1-\xi_i, \quad \text{for all } i \in V_1 \right\}.$$

$$U := \left\{ u \in \{0,1\}^{|V_1|} \colon \sum_{i\in V_1} u_i \leq \Gamma \right\}.$$

$$\min_{x\in\{0,1\}^{|V_1|}} \left\{ \sum_{i\in V_1} f_i x_i + \max_{u\in U} \min_{y\in Y(x,u)} \sum_{i\in V_1}\sum_{j\in V_2} c_{ij} y_{ij} \right\}$$

### 4.2. Solving the separation problem.

4.2.1. *Optimality separation.* The KKT conditions read

$$\text{primal feasibility} = \begin{cases} \sum_{i\in V_1} y_{ij} = 1, & \text{for all } j \in V_2, \\ \sum_{j\in V_2} d_j y_{ij} \leq q_i x_i, & \text{for all } i \in V_1, \\ y_{ij} \leq 1-\xi_i, & \text{for all } i \in V_1, j \in V_2, \\ y_{ij} \geq 0, & \text{for all } i \in V_1, j \in V_2, \end{cases}$$

$$\text{dual feasibility} = \begin{cases} \beta_i, \gamma_{ij} \leq 0, & \text{for all } i \in V_1, j \in V_2, \\ \delta_{ij} \geq 0, & \text{for all } i \in V_1, j \in V_2, \end{cases}$$

$$\text{stationarity} = \begin{cases} \alpha_j + d_j\beta_i + \gamma_{ij} + \delta_{ij} = c_{ij}, & \text{for all } i \in V_1, j \in V_2, \end{cases}$$

$$\text{complementarity} = \begin{cases} \left(\sum_{j\in V_2} d_j y_{ij} - q_i x_i\right)\beta_i \leq 0, & \text{for all } i \in V_1, \\ (y_{ij} - 1 + \xi_i)\,\gamma_{ij} \leq 0, & \text{for all } i \in V_1, j \in V_2, \\ y_{ij}\delta_{ij} \leq 0, & \text{for all } i \in V_1, j \in V_2. \end{cases}$$

Strong duality implies

$$\sum_{j\in V_2} \alpha_j + \sum_{i\in V_1} q_i x_i \beta_i + \sum_{i\in V_1}\sum_{j\in V_2} (1-\xi_i)\,\gamma_{ij} = \sum_{i\in V_1}\sum_{j\in V_2} c_{ij} y_{ij}. \tag{10}$$

Proposition 2. $\alpha \geq 0$.

PROOF. We start with the lower bound. By the stationaty conditions, the following holds.

$$\begin{aligned}
&\alpha_j y_{ij} + d_j \underbrace{\beta_j y_{ij}}_{\leq 0} + \underbrace{\gamma_{ij} y_{ij}}_{\leq 0} + \underbrace{\delta_{ij} y_{ij}}_{=0} = c_{ij} y_{ij}, \quad \text{for all } i \in V_1, j \in V_2,\\
\implies \quad &\alpha_j y_{ij} \geq c_{ij} y_{ij} \geq 0, \quad \text{for all } i \in V_1, j \in V_2,\\
\implies \quad &\sum_{i \in V_1} \alpha_j y_{ij} \geq 0, \quad \text{for all } j \in V_2,\\
\implies \quad &\alpha_j \underbrace{\sum_{i \in V_1} y_{ij}}_{=1} \geq 0, \quad \text{for all } j \in V_2.
\end{aligned}$$

□

4.2.2. *Feasibility separation.*

$$\begin{aligned}
\min_{y,u^+,u^-,v,w} \quad & \sum_{j \in V_2} u_j^+ + u_j^- + \sum_{i \in V_1} v_i + \sum_{i \in V_1} \sum_{j \in V_2} w_{ij}\\
\text{s.t.} \quad & \sum_{i \in V_1} y_{ij} = 1 + u_j^+ - u_j^-, \quad \text{for all } j \in V_2,\\
& \sum_{j \in V_2} d_j y_{ij} \leq q_i x_i + v_i, \quad \text{for all} i \in V_1,\\
& y_{ij} \leq 1 - \xi_i + w_{ij}, \quad \text{for all } i \in V_1, j \in V_2,\\
& y_{ij}, u_j^+, u_j^-, v_i, w_{ij} \geq 0, \quad \text{for all } i \in V_1, j \in V_2.
\end{aligned}$$

4.2.3. *An alternative approach: Farkas separation.*

$$\sum_{i \in V_1} f_i x_i + \sum_{i \in V_1} \sum_{j \in V_2} c_{ij} y_{ij} \leq x_0, \tag{12a}$$

$$\sum_{i \in V_1} y_{ij} = 1, \quad \text{for all } j \in V_2, \tag{12b}$$

$$\sum_{j \in V_2} d_j y_{ij} \leq q_i x_i, \quad \text{for all } i \in V_1, \tag{12c}$$

$$y_{ij} \geq 0, \quad \text{for all } i \in V_1, j \in V_2. \tag{12d}$$

## 5. Robust bilevel problems with wait-and-see followers

## 6. Example: The facility location problem with wait-and-see follower and facility disruption

### 6.1. Solving the separation problem.

6.1.1. *Feasibility separation.*

6.1.2. *Optimality separation.*

CHAPTER 15

# K-adaptability