idol: A C++ Framework for Optimization

Reference Manual

For idol v0.8.2-alpha.

Henri Lefebvre

Contents

Part 1

Mixed-integer Optimization

CHAPTER 1

Modeling a mixed-integer problem

Contents

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:

$$
\begin{aligned}
\min_{x} \quad & c^\top x + x^\top D x + c_0 && \text{(1a)}\\
\text{s.t.} \quad & a_{i\cdot}^\top x + x^\top Q^i x \le b_i, \quad \text{for all } i = 1, \dots, m, && \text{(1b)}\\
& \ell_j \le x_j \le u_j, \quad \text{for all } j = 1, \dots, n, && \text{(1c)}\\
& x_j \in \mathbb{Z}, \quad \text{for all } j \in J \subseteq \{1, \dots, n\}. && \text{(1d)}
\end{aligned}
$$

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

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 $0 \leq \ell \leq u \leq 1$
continuous	$j \notin J$
free	$l = -\infty$ and $u = \infty$
non-negative	$\ell \geq 0$
non-positive	$u \leq 0$
bounded	$-\infty < \ell \leq u < \infty$
fixed	$\ell = u$

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).

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 toy 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.

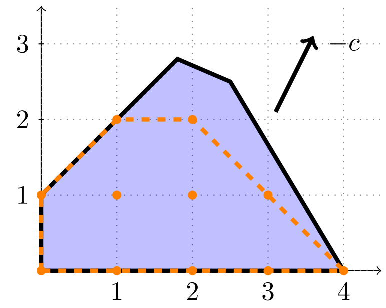

FIGURE 1. Feasible region (shaded in blue) for Problem (2), with integer-feasible points highlighted and objective direction shown.

```cpp
1 #include <iostream>
2 #include "idol/modeling.h"
3
4 using namespace idol;
5
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.

```cpp
    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 2×3 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.

```
    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.

```
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

Contents

1. Optimizers and optimizer factories

2. Example: the binary knapsack problem

3. Interfacing with Cplex

4. Interfacing with GLPK

5. Interfacing with Gurobi

6. Interfacing with HiGHS

7. Interfacing with JuMP

8. Interfacing with Mosek

9. Interfacing with Osi

10. Creating your own optimizer

CHAPTER 3

Callbacks

Contents

1. Universal callbacks

2. Adding user cuts

3. Example: separating knapsack cover inequalities

4. Adding lazy constraints

5. Example: a toy benders decomposition

CHAPTER 4

Writing a custom Branch-and-Bound algorithm

CHAPTER 5

Column Generation and Branch-and-Price

Contents

1. The Dantzig-Wolfe decomposition

1.1. Introduction.

1.2. Dual-smoothing stabilization.

2. Branch-and-Price

3. Example: the generalized assignment problem

4. Strong branching

CHAPTER 6

Penalty alternating direction method

Part 2

Bilevel Optimization

CHAPTER 7

Modeling a bilevel problem

Contents

1. Introduction

2. The Bilevel::Description class

3. Reading and writing instance files from BOBILib

CHAPTER 8

Problems with continuous follower

Contents

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

1. The strong-duality single-level reformulation

2. The KKT single-level reformulation

3. Linearization techniques for the KKT single-level reformulation

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

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

The KKT system reads

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

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

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

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

3.1. Using SOS1 constraints.

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

3.2. Using the big-M approach.

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

4. Example: a toy bilevel problem

5. Min-max problems

6. 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

Contents

1. Interfacing with MibS

2. The extended single-level reformulation approach

3. The penalty alternating direction method

4. Example: the Moore and Bard example

CHAPTER 10

Pessimistic bilevel optimization

Contents

1. Introduction

2. Reformulation as an optimistic bilevel problem

Part 3

Robust Optimization

CHAPTER 11

Modeling a robust problem

Contents

1. Single-stage problems

2. Two-stage problems

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

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

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

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

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

3. Robust 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

Contents

1. Duality-based reformulations

2. Bilevel reformulations

CHAPTER 13

Column-and-constraint generation

Contents

1. Introduction

In this chapter, we dive into the column-and-constraint generation (CCG) algorithm which is a state-of-the-art method for tackling two-stage robust problems. It was first introduced by Zeng and Zhao (2013) in the context of linear two-stage problems and was later on extended to broader settings such as mixed-integer second-stage decisions (see, e.g., Zeng and Zhao 2012, Subramanyam 2022 and Lefebvre and Subramanyam 2025) or convex second-stage constraints (see, e.g, Khademi et al. 2022 and Lefebvre et al. 2025).

In a nutshell, the CCG algorithm iteratively constructs and solves a sequence of restricted master problems which approximates the original problem. To do this, it considers a finite subset of scenarios in place of the—typically infinite—original uncertainty set. Then, by solving a separation problem—the so-called adversarial problem—it identifies a new critical scenario where the current first-stage solution fails to satisfy the second-stage constraints under the full uncertainty set. If such a scenario is found, it is incorporated into the master problem, and the process repeats. Otherwise, the algorithm terminates, having found a robust feasible first-stage decision.

HL

todo

The rest of this chapter is organized as follows...

1.1. Problem statement and assumptions. We consider two-stage robust problems of the form

$$\min_{x\in X} c^\top x + \max_{u\in U} \min_{y\in Y(x,u)} d^\top y \tag{8}$$

where X denotes the set of feasible first-stage decisions, U denotes the uncertainy set and $Y(x,u)$ denotes the set of feasible second-stage decisions given a first-stage decision $x \in X$ and a scenario $u \in U$. In what follows, we consider the fairly general case described by

$$\begin{aligned} X &:= \left\{ x \in \tilde{X} \colon Ax \geq a \right\}, \\ U &:= \left\{ u \in \tilde{U} \colon Fu \leq g \right\}, \\ Y(x,u) &:= \left\{ y \in \tilde{Y} \colon Cx + Dy + E(x)u \geq b \right\}, \end{aligned}$$

where the sets $\tilde{X}$, $\tilde{U}$ and $\tilde{Y}$ are used to impose simple restrictions on the first-stage decisions, the uncertain parameters and the second-stage decisions, respectively. We denote by n_x, n_u and n_y the number of first-stage decisions, uncertain parameters and second-stage decisions, respectively. For instance, with $\tilde{X} = \mathbb{R}^{n_x}$, $\tilde{U} = \mathbb{R}^{n_u}$ and $\tilde{Y} = \mathbb{R}^{n_y}$, Problem (8) is a linear two-stage robust problem.

We make the following assumptions which are sufficient to ensure convergence of the column-and-constraint algorithm.

Assumption 1. *The sets X, U and $Y(x,u)$ are compact, possibly empty, for all $x \in X$ and $u \in U$.*

Assumption 2. *For all $x \in X$, the adversarial problem $\sup_{u\in U} \min_{y\in Y(x,u)} d^\top y$ is either infeasible or attains its supremum.*

One note about these assumptions. They are sufficient conditions to ensure the convergence of the CCG algorithm. If they are not satisfied, the CCG algorithm is not guaranteed to converge and extra care should be taken. Also note that Assumption 2 is *hard* to check at least from an implementation viewpoint. Thus, idol will not check this assumption for you. However, the following proposition holds.

Proposition 1. *Let Assumption 1 be satisfied and assume that $\tilde{Y} = \mathbb{R}^{n_y}$. Then, Assumption 2 holds.*

1.2. The overall procedure. At each iteration $K \geq 1$, the CCG algorithm solves the so-called restricted master problem which consists in solving Problem (8) having replaced the uncertainty set U by a discrete subset of scenarios $\{u^1, \dots, u^K\}$. This problem can be formulated as

$$\begin{aligned} \min_{x\in X, x_0, y^1,\dots,y^K \in \tilde{Y}} \quad & c^\top x + x_0 && \text{(9a)} \\ \text{s.t.} \quad & x_0 \geq d^\top y^k, \quad \text{for all } k = 1,\dots,K, && \text{(9b)} \\ & Cx + Dy^k + E(x)u^k \geq b, \quad \text{for all } k = 1,\dots,K. && \text{(9c)} \end{aligned}$$

Note that vectors u^k with $k = 1,\dots,K$ are inputs of this model. Also note that the algorithm requires (at least) one initial scenario u^1 so that the master problem (9) is lower bounded. Several strategies exist to find this initial scenario and are discussed in Section 6. Also note that if, at any iteration K, the restricted master problem (9) is infeasible, then (8) is infeasible and the algorithm stops.

Now, assume that the restricted master problem (9) is feasible and let a solution be denoted by $(x, x_0, y^1, \dots, y^K)$. We need to check that the point x is feasilbe for Problem (8), i.e., that for all $u \in U$, there exists $y \in Y(x,u)$, otherwise identify a scenario u for which $Y(x,u)$ is empty. This is called the "feasibility separation". If x is a feasible first-stage decision, we need to check that the optimal objective value of the restricted master problem (9) $c^\top x + x_0$ is, indeed, the correct cost associated to x, i.e., that for all $u \in U$, $x_0 = \min\{d^\top y \colon y \in Y(x,u)\}$ holds, otherwise identify a scenario $u \in U$ such that $\min_y\{f^\top y \colon y \in Y(x,u)\} > x_0$. This is called the "optimality separation". We now discuss how these two separation tasks are performed.

In idol, feasibility separation is done by solving the bilevel optimization problem

$$v_{\text{feas}} := \max_{u\in U} \ \min_{y,z} \left\{ e^\top z \colon y \in \tilde{Y},\ z \geq 0,\ Cx + Dy + E(x)u + z \geq b \right\}. \tag{10}$$

It can be easily checked that if $v_{\text{feas}} > 0$, then the upper-level solution $u \in U$ is such that $Y(x,u) = \emptyset$, i.e., x is not a feasible first-stage decision, and the point u is added to the master problem (9) for the next iteration. Otherwise, the algorithm continues with optimality separation (if any). Note that it is not necessary to solve the separation problen to global optimality to show that $v_{\text{feas}} > 0$: any bilevel feasible point would do if its associated upper-level objective value is strictly greater than 0. Internally, Problem (10) is formulated as

$$\begin{aligned} -\min_{u\in U,y,z} \quad & -e^\top z \\ \text{s.t.} \quad & (y,z) \in \arg\min_{\bar{y},\bar{z}} \left\{ e^\top \bar{z} \colon \bar{y} \in \tilde{Y},\ \bar{z} \geq 0,\ Cx + D\bar{y} + E(x)\bar{u} + \bar{z} \geq b \right\}. \end{aligned}$$

Clearly, the two formulations are equivalent on the level of global solutions. This formulation allows idol to rely on any optimizer designed for general bilevel problems, i.e., solvers implementing the Bilevel::SolverInterface interface. Note that this does not prevent a specific solver to exploit the structure of this specific problem such as the zero-sum property or the absence of coupling constraints.

Similary, optimality separation is performed by solving the bilevel problem

$$v_{\text{opt}} := \max_{u\in U} \ \min_{y} \left\{ d^\top y \colon y \in \tilde{Y},\ Cx + Dy + E(x)u \geq b \right\}. \tag{11}$$

Note that this problem is only solved if either feasibility separation has been turned off—e.g., because the problem is known to have complete recourse—or the second-stage problem is feasible for all $u \in U$ given x. Hence, Problem (11) is always feasible. Note that $c^\top x + v_{\text{opt}}$ yields an upper bound on the optimal objective function value of Problem (8). This upper bound is used to show that the algorithm has converged by comparing this upper bound to the lower bound given, at each iteration, by the master problem (9). A scenario u which disproves the optimality of x is always added to the master problem (9) for the next iteration. Otherwise, the algorithm stops with a proof of optimality for x. Just like the feasibility separation problem, the optimality separation problem is internally modeled as the following bilevel problem

$$\begin{aligned} -\min_{u\in U,y} \quad & -d^\top y \\ \text{s.t.} \quad & y \in \arg\min_{\bar{y}} \left\{ d^\top \bar{y} \colon \bar{y} \in \tilde{Y},\ Cx + D\bar{y} + E(x)\bar{u} \geq b \right\}. \end{aligned} \tag{12}$$

Again, this allows idol to rely on general bilevel solvers without restricting the possibility of exploiting its structure. A complete description of the procedure is presented in Algorithm 1.

Algorithm 1 Column-and-constraint generation with two-step separation

...
while **do**
    ...
    if true **then**
        ...
    end if
    ...
end while

As we discussed it so far, separation is always done in a "two-step" manner: first, feasibility is checked, then optimality is checked. Another possibility is to perform a single separation which checks both feasibility and optimality at the same time. We call this separation approach the "joint separation". Given a current first-stage decision x and a second-stage cost estimate x_0, one solves the bilevel problem

$$v_{\text{joint}} := \max_{u\in U} \min_{(y,z_0,z)\in Y^{*+}(x,u)} z_0 + e^\top z,$$

with $Y^*(x,u)$ defined by

$$Y^*(x,u) := \left\{ (y,z_0,z) \in \tilde{Y} \times \mathbb{R}^{m_y+1}_{\geq 0} \colon d^\top y - z_0 \leq x_0,\ Cx + Dy + E(x)u + z \geq b \right\}.$$

Let (u, y, z_0, z) denote a solution to this bilevel problem—$(y, z_0 z)$ being an optimal point of the inner minimization problem given u—if $z > 0$, then the first-stage decision x is not feasible for u. Otherwise ($z = 0$), then $z_0 = d^\top y - x_0$ and $x_0 + v_{\text{joint}}$ is an upper bound on the optimal objective function value of Problem (8). Joint separation was introduced by Ayoub and Poss (2016) for linear problems and was extended to convex problems by Lefebvre et al. (2025). Algorithm 2 presents the overall algorithm.

Algorithm 2 Column-and-constraint generation with joint separation

...
while **do**
    ...
    if true **then**
        ...
    end if
    ...
end while

1.3. The Robust::ColumnAndConstraintGeneration optimizer. We now move on to implementing a column-and-constraint generation algorithm with idol. Thus, the reader should be familiar with modeling two-stage robust problems in idol. If this is not the case, please refer to Chapter 11. Let's start.

In idol, a column-and-constraint generation algorithm can be implemented by using the class Robust::ColumnAndConstraintGeneration which is an optimizer factory. It can be created as follows.

```
auto ccg = Robust::ColumnAndConstraintGeneration(
    robust_description, bilevel_description
);
```

Here, robust_description is an object of the class Robust::Description which contains the uncertainty set as well the uncertain coefficients in the model. The bilevel_description argument is a Bilevel::Description object which describes which variables and constraints are part of the first and second stage. Recall that all "upper-level" variables and constraints are considered to belong to the first stage while all "lower-level" variables and constraints are second-stage entities. Also recall that the lower-level objective function must not be defined here. If a lower-level objective function were to be defined, idol would interpret the problem as a robust bilevel problem with wait-and-see follower instead of a two-stage robust problem, leading to a different algorithmic scheme.

There are two necessary ingredients to be specified for a column-and-constraint generation algorithm to run: how to solve the master problem and how to solve the separation problem. We will dive into the separation problem in the next sections. Let's focus on the master problem. The master problem is a single-level model presented in (9). Thus, we need to specify an optimizer to be used for finding an optimal point at every iteration. This can be achieved with the with_master_optimizer method. In the following example, we use Gurobi.

```
ccg.with_master_optimizer(Gurobi());
```

For now, let's assume that our algorithm is entirely configured and ready to be executed, i.e., assume that a method for solving the separation problem has been configured. Then, with model being an object of the class Model storing the deterministic version of our problem, we can do the following to run the column-and-constraint generation algorithm.

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

That's it! Now the column-and-constraint generation algorithm is being executed.

2. Separation with continuous second stage

In this section, we focus on the specific case in which $\tilde{Y} = \mathbb{R}^{n_y}$ and discuss various techniques to solve the separation problem. To this end, we focus on optimality separation. The feasibility separation and the joint separation can be handled in a much similar way. Note that the separation problem is automatically formulated by idol. Thus, the only input from the user that is required is how to solve the corresponding bilevel problem. Depending on the separation type, this can be done with the add_optimality_separation_optimizer method, the add_feasibility_separation_optimizer method or the add_joint_separation_optimizer method. These methods take only one argument which is an optimizer factory implementing the Bilevel::SolverInterface interface. At each iteration, the appropriate bilevel model is built by idol and the corresponding optimizer factory is used to create an optimizer for solving this bilevel problem.

Note that a separation phase is only triggered if at least one optimizer for this phase has been specified. For instance, if a problem is known to have complete recourse—meaning that the second-stage problem is always feasible—then not specifying an optimizer for the feasibility separation will produce a column-and-constraint generation algorithm which does not check for feasibility. Also note that more than one optimizer per separation type can be specified. Hence, heuristic approaches can be added too. The last optimizer should always be an exact solver to prove convergence of the method.

We end with a simple example which prepares the column-and-constraint algorithm to solve the optimality separation problems by means of a KKT single-level reformulation which linearizes the complementarity constraints by means of SOS1 constraints.

```
auto kkt = Bilevel::KKT();
kkt.with_sos1_constraints(true);

ccg.add_optimality_separation_optimizer(kkt + Gurobi());
```

As you can see, it is very easy to configure. For more details on bilevel optimizers, the reader my refer to the section on bilevel optimization.

We now give a more detailed description on how to perform optimality separation using different approaches. Recall that optimality separation is only performed if $Y(x,u) \neq \emptyset$ for all $u \in U$, i.e., after feasibility separation or if it is known that $Y(x,u) \neq \emptyset$ for all $x \in X$ and all $u \in U$. Hence, the separation problem (11)—which, internally is modeled as (12)—is always feasible.

2.1. KKT single-level reformulation. Recall that the second-stage primal problem:

$$\min_y \left\{ f^\top y \colon Cx + Dy + E(x)u \geq b \right\}.$$

Because it is linear, the KKT conditions are both necessary and sufficient for optimality. Hence, any point (y,λ) is a primal-dual solution to the second-stage problem if and only if it satisfies its associated KKT system

$$Cx + Dy + E(x)u \geq b, \quad D^\top \lambda = d, \quad \lambda \geq 0, \quad (Cx + Dy + E(x)u - b)^\top \lambda = 0.$$

Thus, one can equivalently replace the lower-level problem in the separation problem (11) by its KKT conditions, leading to the nonlinear problem

$$\begin{aligned}
\max_{u,y,\lambda} \quad & d^\top y && \text{(13a)}\\
\text{s.t.} \quad & u \in U, && \text{(13b)}\\
& Cx + Dy + E(x)u \geq b, && \text{(13c)}\\
& D^\top \lambda = d, \quad \lambda \geq 0, && \text{(13d)}\\
& (Cx + Dy + E(x)u - b)^\top \lambda = 0. && \text{(13e)}
\end{aligned}$$

This problem is a bilinear optimization problem which can be solved using standard nonlinear techniques. To implement this method, simply call the add_optimality_separation_optimizer with an optimizer factory of the class Bilevel::KKT. The following code snippet shows you how it's done.

```
auto kkt = Bilevel::KKT();
ccg.add_optimality_separation_optimizer(kkt + Gurobi());
```

Here, idol will automatically create this optimizer at every iteration to solve the separation problem. This optimizer then automatically performs a KKT single-level reformulation of the corresponding bilevel problem and solves it with Gurobi.

An alternative approach is to linearize constraints (13e) by introducing auxiliary binary variables. However, this approach requires to compute bounds on the dual variables λ which, to the best of our knowledge, cannot be done efficiently without exploiting problem-specific knowledge. Nevertheless, this additional work has been shown to be benificial since solving the resulting mixed-integer linear problem is typically much easier than solving the nonlinear problem. Thus, assume that we know that there always exists dual solutions satisfying $\lambda \leq M$ for some diagonal matrix $M \geq 0$. Then, the complementarity constraints (13e) can be replaced by

$$0 \leq \lambda \leq Mz, \quad 0 \leq Cx + Dy + E(x)u - b \leq M \circ (e - z), \quad z \in \{0,1\}^{m_y}.$$

Replacing constraints (13e) by these new constraints leads to a mixed-integer linear problem. To implement this, one needs to provide bounds to idol that will, then, use them to perform the linearization. For a complete description, we refer to Chapter 9 and the related section on bound providers. Now, assuming that bound_provider is an object of a child class of Reformulators::KKT::BoundProvider, one implements this methods as follows.

```
auto kkt = Bilevel::KKT();
kkt.with_bound_provider(bound_provider);
ccg.add_optimality_separation_optimizer(kkt + Gurobi());
```

In the next section, a complete example is given to solve an uncapacitated facility location problem with facility disruption.

One important note. In practice, choosing a wrong value for M may result in a column-and-constraint generation scheme that is no longer convergent. Hence, it is necessary that the bounds M be valid. On the contrary, choosing a too large value for M leads to bad performance in terms of computational time and/or numerical stability. Another approach to tackle the nonlinear constraints (13e) that does not require computing big-M values is through SOS1 constraints. Again, we refer to Chapter 9 for more details. Here is how it can be implemented.

```
auto kkt = Bilevel::KKT();
kkt.with_sos1_constraints(true);
ccg.add_optimality_separation_optimizer(kkt + Gurobi());
```

2.2. Strong-duality single-level reformulation. Similar to the KKT-based separation, one can exploit the strong-duality single-level reformulation of the bilevel separation problem. This model reads

$$\begin{aligned} \max_{u,y,\lambda} \quad & d^\top y \\ \text{s.t.} \quad & u \in U, \\ & Cx + Dy + E(x)u \geq b, \\ & D^\top \lambda = d, \quad \lambda \geq 0, \\ & f^\top y \leq (b - Cx - E(x)u)^\top \lambda. \end{aligned}$$

This model can be solved using a nonlinear optimization solver like Gurobi. Implementing this technique is straightforward with the Bilevel::StrongDuality optimizer factory.

```
auto strong_duality = Bilevel::StrongDuality();
ccg.add_optimality_separation_optimizer(
    strong_duality + Gurobi()
);
```

2.3. Dualized second-stage problem. Another approach for solving the separation problem is to exploit linear duality and the max-min structure of the separation problem. Recall that the second-stage problem is feasible for all $u \in U$ given x. Hence, the lower-level problem in (11) is feasible and bounded and attains the same optimal objective function value as its dual problem. The dual reads

$$\max_{\lambda} \left\{ (b - Cx - E(x)u)^\top \lambda \colon D^\top \lambda = d, \ \lambda \geq 0 \right\}.$$

Replacing the lower-level problem by its dual in (11) leads to the nonlinear model

$$\max_{u \in U, \lambda} \left\{ (b - Cx - E(x)u)^\top \lambda \colon D^\top \lambda = d, \ \lambda \geq 0 \right\} \tag{14}$$

which can be solved by standard nonlinear approaches. To implement this technique, we will make use of the Bilevel::MinMax::Dualize optimizer factory. The following code snippet gives you an easy-to-read example.

```
auto dualize = Bilevel::MinMax::Dualize();
ccg.add_optimality_separation_optimizer(dualize + Gurobi());
```

HL

Not yet implemented.

2.4. Dualized second-stage problem with KKT reformulation.

Building upon the duality-based separation, consider problem (14) and let us write it as

$$\begin{aligned} \max_{\lambda} \quad & (b - Cx)^\top \lambda + \max_{u \in U} -\lambda^\top E(x)^\top u \\ \text{s.t.} \quad & D^\top \lambda = d, \quad \lambda \geq 0. \end{aligned}$$

Note that the inner maximization problem is feasible and bounded since U is nonempty and compact. Hence, we may replace it by its KKT conditions

$$Fu \leq g, \quad \mu \geq 0, \quad F^\top \mu = -E(x)^\top \lambda, \quad \mu^\top (Fu - g) = 0.$$

The resulting formulation for the separation problem is then the nonlinear problem

$$\begin{aligned} \max_{u \in U, \lambda, \mu} \quad & (b - Cx)^\top \lambda + g^\top \mu \\ \text{s.t.} \quad & D^\top \lambda = d, \quad \lambda \geq 0, \\ & \mu \geq 0, \quad F^\top \mu = -E(x)^\top \lambda, \quad \mu^\top (Fu - g) = 0. \end{aligned}$$

Here again, the nonlinear terms arising from the complementarity constraints of the KKT conditions can be linearized using auxiliary binary variables and valid bounds on the dual variables μ. In such a case, the resulting problem is a mixed-integer linear problem.

3. Example: the uncapacitated facility location problem

We consider an uncapacitated facility location problem (UFLP) in which facilities are subject to uncertain disruptions as studied in Cheng et al. (2021). To that end, let V_1 be a set of facilities location and let V_2 be a set customers. For each facility $i \in V_1$, we let f_i denote the opening cost and q_i its capacity. Each customer $j \in V_2$ is associated to a given demand d_j and a marginal penalty for unmet demand p_j. Each connection $(i,j) \in V_1 \times V_2$ has a unitary transportation cost noted c_{ij}. The deterministic uncapacitated facility location problem can be modeled as

$$\begin{aligned} \min_{x,y,z} \quad & \sum_{i \in V_1} f_i x_i + \sum_{i \in V_1} \sum_{j \in V_2} c_{ij} d_j y_{ij} + \sum_{j \in V_2} p_j d_j z_j && \text{(17a)} \\ \text{s.t.} \quad & \sum_{i \in V_1} y_{ij} + z_j = 1, \quad \text{for all } j \in V_2, && \text{(17b)} \\ & y_{ij} \leq x_i, \quad \text{for all } i \in V_1, \text{for all } j \in V_2, && \text{(17c)} \\ & y_{ij} \geq 0, z_j \geq 0, \quad \text{for all } i \in V_1, j \in V_2, && \text{(17d)} \\ & x_i \in \{0,1\}, \quad \text{for all } i \in V_1. && \text{(17e)} \end{aligned}$$

The uncertain ingredient of the problem under consideration is that some facilities can be made unavaible. If this is the case, we say that a given facility is disrupted. We consider the binary budgeted knapsack uncertainty set

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

where, for all facility $i \in V_1$, $u_i = 1$ if and only if faciltiy i is disrupted. The parameter Γ controls the maximum number of facilities which can be disrupted at the same time and is part of the model. As it typically occurs in real-world applications, we assume that facilities have to be planned before any disruption can be anticipated while deciding the operational decisions of serving customers from facilities can be delayed at a later instant, where disrupted facilities are known. Hence, the two-stage robust problem reads

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

where the second-stage feasible set $Y(x,u)$ is defined for a given first-stage decision $x \in \{0,1\}^{|V_1|}$ and a given scenario $u \in U$ as

$$Y(x,u) := \left\{ y \in \mathbb{R}^{|V_1|\times|V_2|} : \text{(17b)–(17d) and } y_{ij} \le 1 - u_i, \quad \text{for all } i \in V_1 \right\}.$$

The goal of this example is to show how to implement a CCG algorithm to solve this problem. To do this, we first need to describe how the adversarial problem can be solved. This is the subject of the next section.

3.1. Modeling the robust UFLP with facility disruption in idol. As presented in Chapter 11, we first need to create the deterministic model (17). To this end, we will use the method Problems::FLP::read_instance_2021_Cheng_et_al(const std::string&) to read an instance file from Cheng et al. (2021). Such instances can be found at https://drive.google.com/drive/folders/1Gy_guJIuLv52ruY89m4Tgrz49FiMspzn?usp=sharing. With this, we can read an instance as follows.

```
const auto instance = Problems::FLP::read_instance_2021_Cheng_et_al("/path/to/instance.txt");
const unsigned int n_customers = instance.n_customers();
const unsigned int n_facilities = instance.n_facilities();
```

The deterministic model is rather straightforward to build.

```
Env env;
Model model(env);

// Create variables
const auto x = model.add_vars(Dim<1>(n_facilities),
                              0, 1, Binary, 0, "x");
const auto y = model.add_vars(Dim<2>(n_facilities, n_customers),
                              0, Inf, Continuous, 0, "y");
const auto z = model.add_vars(Dim<1>(n_customers),
                              0, Inf, Continuous, 0, "z");

// Create assignment constraints
for (auto j : Range(n_customers)) {
    auto lhs = idol_Sum(i, Range(n_facilities), y[i][j]) + z[j];
    model.add_ctr(lhs <= instance.capacity(i));
}

// Create activation constraints
for (auto i : Range(n_facilities)) {
    for (auto j : Range(n_customers)) {
        model.add_ctr(y[i][j] <= x[i]);
    }
}

// Create objective function
auto objective =
    idol_Sum(i, Range(n_facilities),
        instance.fixed_cost(i) * x[i] +
        idol_Sum(j, Range(n_customers),
```

```
                    instance.per_unit_transportation_cost(i,j) * instance.demand(j) * y[i][j]
                )
            ) +
            idol_Sum(j, Range(n_customers),
                instance.per_unit_penalty(j) * instance.demand(j) * z[j]
            );
    model.set_obj_expr(std::move(objective));
```

Next, we need to declare the two-stage structure, i.e., describe which variables and which constraints are part of the second-stage problem. This is done through the Bilevel::Description class. By default, all variables and constraints are defined as first-stage variables and constraints. Here, the second-stage variables are y and z while all constraints are second-stage constraints. Hence, the following code snippet.

```
    Bilevel::Description bilevel_description(env);

    // Make y and z second-stage variables
    for (auto j : Range(n_customers)) {
        bilevel_description.make_lower_level(z[j]);
        for (auto i : Range(n_facilities)) {
            bilevel_description.make_lower_level(y[i][j]);
        }
    }

    // Make all constraints second-stage constraints
    for (const auto& ctr : model.ctrs()) {
        bilevel_description.make_lower_level(ctr);
    }
```

To end our modeling of the robust UFLP, we need to define two more things: the uncertainty set and the uncertain coefficients in the second-stage. Let's start with the uncertainty set. Here, we use $\Gamma = 2$.

```
    Model uncertainty_set(env);
    const double Gamma = 2;
    const auto u = uncertainty_set.add_vars(Dim<1>(n_facilities),
                                            0, 1, Binary, 0, "u");
    uncertainty_set.add_ctr(
        idol_Sum(i, Range(n_facilities), u[i]) <= Gamma
    );
```

Finally, we need to describe where these parameters appears in the deterministic model, i.e., where does u impact the deterministic model. We do this through the Robust::Description class. To do this, we will first need to identify the constraints which are uncertain, i.e., the activation constraints (17c). We can do so, e.g., by relying on the indices of the constraints within the model we just created. Indeed, we know that the constraint "$y_{ij} \le x_i$" has an index equal to $|V_1| + i|V_2| + j$ for all $i \in V_1$ and all $j \in V_2$. Another way could have been to store these constraints in a separate container or to rely on the constraints' name. In the uncertain version, the activation constraint "$y_{ij} \le x_i$" is changed by adding the term "$-x_i u_i$" to it. Hence, the following code snippet.

```
    Robust::Description robust_description(uncertainty_set);
    for (auto i : Range(n_facilities)) {
        for (auto j : Range(n_customers)) {

          // Get activation constraint (i,j)
          const auto index = n_customers + i * n_customers + j;
          const auto& c = model.get_ctr_by_index(index);

```

```
            // Add uncertain term
            robust_description.set_uncertain_mat_coeff(c, x[i], u[i]);
        }
    }
```

In a nutshell, the call to Robust::Description::set_uncertain_mat_coeff tells idol that an uncertain coefficient for x should be added and equals u_i, i.e.,

$$y_{ij} - x_i \le 0 \quad \longrightarrow \quad y_{ij} - x_i + x_i u_i \le 0.$$

That's it, our robust UFLP is now completely modeled in idol.

3.2. Preparing the column-and-constraint optimizer. We are now ready to create our optimizer for solving problem (17). Recall that the optimizer has an optimizer factory called Robust::ColumnAndConstraintGeneration which can be used as follows.

```
auto ccg = Robust::ColumnAndConstraintGeneration(
                    robust_description,
                    bilevel_description
                );
```

As you can see, it is necessary to provide both the bilevel description—so that idol knows which variables and constraints are in the first- or second-stage—, and the robust description—so that uncertain coefficients as well as the uncertainty set are also known. When we are done configuring the CCG algorithm, we will be able to call the Model::use method to set up the optimizer factory and the Model::optimize method to solve the problem.

```
// Once we are done configuring ccg
model.use(ccg);
model.optimize();
```

Before we can do so, we need to at least give some information on how to solve each optimization problems that appear as a sub-problem in the column-and-constraint algorithm. There are essentially two types of problems to solve: the master problem and the adversarial problem. For the master problem, we will simply use Gurobi. We do this through the following code.

```
ccg.with_master_optimizer(Gurobi());
```

Then, we need to describe how to solve the separation problem. This is the subject of the next section.

3.3. Solving the separation problem. During the CCG algorithm, at every iteration, the master problem is solved. If the master problem is infeasible, we know that the original two-stage robust problem is infeasible. Otherwise, let $(x, y^1, \dots, y^k)$ for some k corresponding to the number of scenarios present in the master problem. Given this point, and a current estimate on the second-stage costs x_0, we need to show that either x is a feasible first-stage decision with a second-stage cost no more than x_0, or exhibit a scenario $\hat{u} \in U$ such that either $Y(x, \hat{u}) = \emptyset$ or the best second-stage decision y^* given x and $\hat{u}$ is such that $d^\top y^* > x_0$. Note that, in the case of the UFLP, the second-stage problem is always feasible. Thus, we "only" need to check that x is an optimal first-stage decision. In what follows, we discuss how to implement this separation for the UFLP.

3.3.1. *Dualized second-stage problem.* The first approach is the well-known duality-based approach which consists in replacing the second-stage primal problem by its dual and linearize products between dual and uncertain variables in the objective function. Let's see how it's done. First, since the second-stage problem is always feasible and bounded, the primal problem attains the same objective value as its dual. Hence, the primal second-stage problem can be replaced by its dual problem

$$\begin{aligned}
\max_{\alpha,\beta,\gamma,\delta} \quad & \sum_{j\in V_2} \alpha_j + \sum_{i\in V_1}\sum_{j\in V_2} x_i(1-u_i)\beta_{ij} && \text{(18a)}\\
\text{s.t.} \quad & \alpha_j + \beta_{ij} + \gamma_{ij} = c_{ij}d_j, \quad \text{for all } i\in V_1, j\in V_2, && \text{(18b)}\\
& \alpha_j + \delta_j = p_j d_j, \quad \text{for all } j \in V_2, && \text{(18c)}\\
& \alpha_j \in \mathbb{R}, \beta_{ij} \le 0, \gamma_{ij} \ge 0, \delta_j \ge 0, \quad \text{for all } i\in V_1, j \in V_2. && \text{(18d)}
\end{aligned}$$

Hence, the separation problem reads

$$\begin{aligned}
\max_{u,\alpha,\beta,\gamma,\delta} \quad & \sum_{j\in V_2} \alpha_j + \sum_{i\in V_1}\sum_{j\in V_2} x_i(1-u_i)\beta_{ij}\\
\text{s.t.} \quad & u \in U, \text{(18b)–(18d)}.
\end{aligned}$$

To implement this technique in idol, we can use the Bilevel::MinMax::StrongDuality optimizer factory. It can be configured as follows.

```
auto duality_based = Bilevel::MinMax::StrongDuality();
duality_based.with_optimizer(Gurobi());

ccg.add_optimality_separation_optimizer(duality_based);
```

By doing so, the optimizer factory duality_based will be called every time a new separation problem needs to be solved. Note that, as such, we did not provide any bounds on the dual variables. Hence, the dualized model will be solved as a nonlinear problem by Gurobi; see also Chapter 9 on bilevel problems with continuous lower-level problems. However, computational experiments have shown that linearizing those terms is beneficial whenever possible. Hence, we now show how such bounds can be passed and exploited by idol.

It is shown in appendix B.1 of Cheng et al. (2021) that a dual solution always exists with $\beta_{ij} \ge \min\{0, d_j(c_{ij} - p_j)\} =: M_{ij}$. Thus, we can linearize the products $w_{ij} := u_i\beta_{ij}$ by introducing binary variables v_{ij} such that

$$M_{ij}v_{ij} \le w_{ij} \le 0, \quad \beta_{ij} \le w_{ij} \le \beta_{ij} - M_{ij}(1-v_{ij}), \quad v_{ij} \in \{0,1\}, \tag{19}$$

for all $i \in V_1$ and all $j \in V_2$. Note that idol can do this automatically. To use this feature, we simply need to provide these bounds to idol. As discussed in Chapter 9, this can be done by means of a child class of the Reformulators::KKT::BoundProvider class which will return the necessary bounds. Here is one possible implementation which relies on a map between the constraints' names and their corresponding bounds on dual variables. Note that the only bounds which need to be returned in this case are those on β since β is the only variable involved in a product. For that reason, only the get_ctr_dual_lb method is implemented (β is known to be non-negative).

```
class UFLPBoundProvider
    : public idol::Reformulators::KKT::BoundProvider {
      idol::Map<std::string, double> m_ctr_dual_bounds;
public:

    // Constructor which computes bounds on the dual variables
    // and fills the m_ctr_dual_bounds map.
    UFLPBoundProvider(
        const idol::Problems::FLP::Instance& t_instance,
        const idol::Vector<idol::Ctr, 2>& t_activation_ctrs) {

        compute_bounds(t_instance, t_activation_ctrs);
```

```

        }

        double get_ctr_dual_lb(const idol::Ctr &t_ctr) override {
            return m_ctr_dual_bounds.at(t_ctr.name());
        }

        double get_ctr_dual_ub(const idol::Ctr &t_ctr) override {
            throw idol::Exception("This method will not be called");
        }

        double get_ctr_slack_lb(const idol::Ctr &t_ctr) override {
            throw idol::Exception("This method will not be called");
        }

        double get_ctr_slack_ub(const idol::Ctr &t_ctr) override {
            throw idol::Exception("This method will not be called");
        }

        double get_var_lb_dual_ub(const idol::Var &t_var) override {
            throw idol::Exception("This method will not be called");
        }

        double get_var_ub_dual_lb(const idol::Var &t_var) override {
            throw idol::Exception("This method will not be called");
        }

        double get_var_ub(const idol::Var &t_var) override {
            throw idol::Exception("This method will not be called");
        }

        [[nodiscard]] BoundProvider *clone() const override {
            return new UFLPBoundProvider(*this);
        }

        void compute_bounds(
            const idol::Problems::FLP::Instance& t_instance,
            const idol::Vector<idol::Ctr, 2>& t_activation_ctrs) {

            const auto n_facilities = t_instance.n_facilities();
            const auto n_customers = t_instance.n_customers();

            // Compute bounds for the activation constraints
            for (auto i : idol::Range(n_facilities)) {
                for (auto j : idol::Range(n_customers)) {
                    const auto& name=t_activation_ctrs[i][j].name();
                    m_ctr_dual_bounds[name] = std::min(
                            0.,
                            t_instance.demand(j) * (t_instance.↩
                                per_unit_transportation_cost(i, j) −↩
                                 t_instance.per_unit_penalty(j) )
                    );
                }
            }

        }
    };
```

The complete code for configuring the CCG algorithm reads as follows.

```
auto ccg = Robust::ColumnAndConstraintGeneration(
                        robust_description,
                        bilevel_description
```

```
                       );

    ccg.with_master_optimizer(Gurobi());

    auto dualize = Bilevel::MinMax::StrongDuality();
    duality_based.with_bound_provider(
        UFLPBoundProvider(instance, activations_constraints)
    );

    ccg.add_optimality_separation_optimizer(dualize + Gurobi());

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

With this code, the dualized model is now being linearized before being solved by Gurobi, i.e., it is solved as a mixed-integer linear problem. Note that there is also a third way to solve the dualized model which is by means of SOS1 constraints. This can be implemented by using the following code.

```
    auto duality_based = Bilevel::MinMax::StrongDuality();
    duality_based.with_sos1_constraints();
```

3.3.2. *KKT-based separation.* Another way to solve the separation problem is to exploit the KKT optimality conditions of the second-stage problem. These conditions are necessary and sufficient for a primal-dual point to be optimal for the second-stage primal and dual problems. These conditions are stated as

$$
\begin{aligned}
\text{primal feasibility} &= \begin{cases} \sum_{i\in V_1} y_{ij} + z_j = 1, & \text{for all } j \in V_2, \\ y_{ij} \le x_i(1-u_i), & \text{for all } i \in V_1, \text{for all } j \in V_2, \\ y_{ij} \ge 0, z_j \ge 0, & \text{for all } i \in V_1, j \in V_2, \end{cases} \\
\text{dual feasibility} &= \begin{cases} \alpha_j + \beta_{ij} + \gamma_{ij} = c_{ij} d_j, & \text{for all } i \in V_1, j \in V_2, \\ \alpha_j + \delta_j = p_j d_j, & \text{for all } j \in V_2, \end{cases} \\
\text{stationarity} &= \Big\{ \alpha_j \in \mathbb{R}, \beta_{ij} \le 0, \gamma_{ij} \ge 0, \delta_j \ge 0, \quad \text{for all } i \in V_1, j \in V_2. \\
\text{complementarity} &= \begin{cases} \beta_{ij}(y_{ij} - x_i(1-u_i)) = 0, & \text{for all } i \in V_1, \text{for all } j \in V_2, \\ \gamma_{ij} y_{ij} = 0, & \text{for all } i \in V_1, \text{for all } j \in V_2, \\ \delta_j z_j = 0, & \text{for all } i \in V_1, \text{for all } j \in V_2. \end{cases}
\end{aligned}
\tag{20}
$$

Hence, the separation problem can be formulated as

$$
\max_{u,y,\alpha,\beta,\gamma,\delta} \left\{ d^\top y \colon u \in U, (20) \right\}.
$$

To implement this technique in idol, we can use the Bilevel::KKT optimizer factory. It can be configured as follows.

```
    auto kkt = Bilevel::KKT();
    kkt.with_optimizer(Gurobi());

    ccg.add_optimality_separation_optimizer(kkt);
```

By doing so, the optimizer factory kkt will be called every time a new separation problem needs to be solved. Note that, as such, we did not provide any bounds on the dual variables. Hence, the KKT reformulation will be solved as a nonlinear problem by Gurobi. However, it is well-known that linearizing the complementarity constraints with binary variables yields much better performance. Hence, we now show how such bounds can be passed and exploited by idol.

Recall from the previous section that there always exists a dual solution such that $\beta_{ij} \ge \min\{0, d_j(c_{ij} - p_j)\}$. From that, we easily show that there exists a dual solution satisfying

$$
\begin{aligned}
&0 \leq \alpha_j \leq p_j d_j, \quad \text{for all } j \in V_2,\\
&0 \geq \beta_{ij} \geq \min\{0, d_j(c_{ij} - p_j)\}, \quad \text{for all } i \in V_1, \text{for all } j \in V_2,\\
&0 \leq \gamma_{ij} \leq c_{ij} d_j + \max\{0, d_j(p_j - c_{ij})\}, \quad \text{for all } i \in V_1, \text{for all } j \in V_2,\\
&0 \leq \delta_j \leq p_j d_j, \quad \text{for all } j \in V_2.
\end{aligned}
$$

We also have the trivial bounds

$$
\begin{aligned}
&0 \leq y_{ij} \leq 1, \quad \text{for all } i \in V_1, j \in V_2,\\
&0 \leq z_j \leq 1, \quad \text{for all } j \in V_2,\\
&0 \leq x_i(1 - u_i) - y_{ij} \leq 1, \quad \text{for all } i \in V_1, \text{for all } j \in V_2.
\end{aligned}
$$

With these, the complementarity constraints can be linearized by means of binary variables. In the following code snippet, we enrich our implementation of the UFLPBoundProvider class so that it returns bounds for all dual variables.

```cpp
class UFLPMasterBoundProvider
    : public idol::Reformulators::KKT::BoundProvider {
    idol::Map<std::string, double> m_ctr_dual_bounds;
    idol::Map<std::string, double> m_ctr_slack_bounds;
    idol::Map<std::string, double> m_var_bound_dual_bounds;

public:
    UFLPMasterBoundProvider(
        const Instance& t_instance,
        const idol::Vector<idol::Var, 2>& t_y,
        const idol::Vector<idol::Var, 1>& t_z,
        const idol::Vector<idol::Ctr, 1>& t_assignment_ctrs,
        const idol::Vector<idol::Ctr, 2>& t_activation_ctrs) {

        compute_bounds(t_instance,
                       t_y,
                       t_z,
                       t_assignment_ctrs,
                       t_activation_ctrs);

    }

    double get_ctr_dual_lb(const idol::Ctr &t_ctr) override {
        return m_ctr_dual_bounds.at(t_ctr.name());
    }

    double get_ctr_dual_ub(const idol::Ctr &t_ctr) override {
        return m_ctr_dual_bounds.at(t_ctr.name());
    }

    double get_ctr_slack_lb(const idol::Ctr &t_ctr) override {
        return m_ctr_slack_bounds.at(t_ctr.name());
    }

    double get_ctr_slack_ub(const idol::Ctr &t_ctr) override {
        return m_ctr_slack_bounds.at(t_ctr.name());
    }

    double get_var_lb_dual_ub(const idol::Var &t_var) override {
        return m_var_bound_dual_bounds.at(t_var.name());
    }

    double get_var_ub_dual_lb(const idol::Var &t_var) override {
        return m_var_bound_dual_bounds.at(t_var.name());
```

```
        }

        double get_var_ub(const idol::Var &t_var) override {
            return 1;
        }

        [[nodiscard]] BoundProvider *clone() const override {
            return new UFLPMasterBoundProvider(*this);
        }

        void compute_bounds(const Instance& t_instance,
                            const idol::Vector<idol::Var, 2>& t_y,
                            const idol::Vector<idol::Var, 1>& t_z,
                            const idol::Vector<idol::Ctr, 1>& ↩
                                t_assignment_ctrs,
                            const idol::Vector<idol::Ctr, 2>& ↩
                                t_activation_ctrs) {

            const auto n_facilities = t_instance.n_facilities();
            const auto n_customers = t_instance.n_customers();

            // Compute bounds for the assignment constraints
            for (auto j : idol::Range(n_customers)) {
                const auto name = t_assignment_ctrs[j].name();
                m_ctr_dual_bounds[name] = t_instance.↩
                    per_unit_penalty(j) * t_instance.demand(j);
                m_ctr_slack_bounds[name] = 1;
            }

            // Compute bounds for the activation constraints
            for (auto i : idol::Range(n_facilities)) {
                for (auto j : idol::Range(n_customers)) {
                    const auto name= t_activation_ctrs[i][j].name();
                    m_ctr_dual_bounds[name] = std::min(0.,
                        t_instance.demand(j) * (
                        t_instance.per_unit_transportation_cost(i,j)
                        - t_instance.per_unit_penalty(j)
                        )
                    );
                    m_ctr_slack_bounds[name] = -1;
                }
            }

            // Compute bounds for the y variables
            for (auto i : idol::Range(n_facilities)) {
                for (auto j : idol::Range(n_customers)) {
                    const auto name = t_y[i][j].name();
                    m_var_bound_dual_bounds[name] =
                        t_instance.per_unit_transportation_cost(i,j)
                        * t_instance.demand(j)
                        + std::max(0.,
                        t_instance.demand(j) * (
                        t_instance.per_unit_penalty(j) -
                        t_instance.per_unit_transportation_cost(i,j)
                        )
                        );
                }
            }

            // Compute bounds for the z variables
            for (auto j : idol::Range(n_customers)) {
                const auto name = t_z[j].name();
```

```
                m_var_bound_dual_bounds[name] = t_instance.↩
                    per_unit_penalty(j)
                    * t_instance.demand(j);
            }

        }

    };
```

Arguably, the core of this approach lies in the compute_bounds method which effectively computes the bounds on all dual variables and slacks to fill the corresponding map. Then, each get_X method returns the corresponding bound stored in the map.

The complete code for configuring the CCG algorithm reads as follows.

```
auto ccg = Robust::ColumnAndConstraintGeneration(
                        robust_description,
                        bilevel_description
                    );

ccg.with_master_optimizer(Gurobi());

auto kkt = Bilevel::KKT();
kkt.with_bound_provider(UFLPBoundProvider(
    instance,
    y,
    z,
    assignment_constraints,
    activation_constraints));

ccg.add_optimality_separation_optimizer(kkt + Gurobi());

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

With this code, the KKT single-level reformulation is now being linearized before being solved by Gurobi, i.e., it is solved as a mixed-integer linear problem. Note that there is also a third way to solve the KKT reformulation which is by means of SOS1 constraints. This can be implemented by using the following code.

```
auto ktk = Bilevel::KKT();
kkt.with_optimizer(Gurobi());
kkt.with_sos1_constraints();
```

3.4. Heuristic separation with PADM.

HL: todo

4. Separation with mixed-integer second stage

We now consider the general case in which $\tilde{Y} = \mathbb{R}^{p_y} \times \mathbb{Z}^{n_y - p_y}$. Note that if $\tilde{U}$ is mixed-integer, then Assumption 2 may not be satisfied. Hence, we only consider discrete uncertainty sets. Similarly to Section 2, we only discuss optimality separation in details. Feasibility and joint separation can be handled in a similar way.

4.1. The bilevel solver MibS. A very simple approach to solve the separation problem is to rely on the external mixed-integer bilevel solver MibS from the coin-or project (Tahernejad et al. 2020). To implement this, one can simply use the optimizer factory Bilevel::MibS. Here is a code snippet to demonstrate how to configure optimality separation with MibS.

```
ccg.add_optimality_separation_optimizer(Bilevel::MibS());
```

Recall that MibS can also internally use IBM CPLEX for checking bilevel feasibility of a given point throughout the branch-and-bound search instead of SYMPHONY. To activate this, use the with_cplex_for_feasibility method as follows.

```
auto mibs = Bilevel::MibS();
mibs.with_cplex_for_feasibility(true);

ccg.add_optimality_separation_optimizer(mibs);
```

4.2. Nested column-and-constraint generation. Subramanyam (2022), Lefebvre and Subramanyam (2025).

HL

Not yet implemented.

5. Example: the capacitated facility location problem

6. Initializing the scenario pool

6.1. Min and max initialization.

HL: todo

6.2. User scenarios.

HL: todo

6.3. The zero-th iteration.

HL: Omitting epigraph to get one $x \in X$ and separate.

7. Robust bilevel problems with wait-and-see followers

8. Example: the uncapacitated facility location problem continued

8.1. Solving the separation problem.

8.1.1. *Optimality separation.*

CHAPTER 14

Objective uncertainty

CHAPTER 15

Heuristic approaches

Contents

1. Affine decision rules

2. K-adaptability

Bibliography

Ayoub, J. and M. Poss (2016). "Decomposition for adjustable robust linear optimization subject to uncertainty polytope." In: *Computational Management Science* 13.2, pp. 219–239. DOI: 10.1007/s10287-016-0249-2.

Cheng, C., Y. Adulyasak, and L.-M. Rousseau (2021). "Robust Facility Location Under Disruptions." In: *INFORMS Journal on Optimization* 3.3, pp. 298–314. DOI: 10.1287/ijoo.2021.0054. URL: http://dx.doi.org/10.1287/ijoo.2021.0054.

Khademi, A., A. Marandi, and M. Soleimani-damaneh (2022). *A New Dual-Based Cutting Plane Algorithm for Nonlinear Adjustable Robust Optimization*. https://optimization-online.org/?p=19112. Optimization Online, Published: 2022/07/19, Updated: 2025/01/08.

Lefebvre, H., E. Malaguti, and M. Monaci (2025). "Exact Approaches for Convex Adjustable Robust Optimization." In: *Optimization Online*. Published: 2022/11/04, Updated: 2025/04/18. URL: https://optimization-online.org/?p=20869.

Lefebvre, H. and A. Subramanyam (2025). *Correction to: A Lagrangian dual method for two-stage robust optimization with binary uncertainties*. arXiv: 2411.04307 [math.OC]. URL: https://arxiv.org/abs/2411.04307.

Subramanyam, A. (2022). "A Lagrangian dual method for two-stage robust optimization with binary uncertainties." In: *Optimization and Engineering* 23.4, pp. 1831–1871. DOI: 10.1007/s11081-022-09710-x.

Tahernejad, S., T. K. Ralphs, and S. T. DeNegre (2020). "A branch-and-cut algorithm for mixed integer bilevel linear optimization problems and its implementation." In: *Mathematical Programming Computation* 12.4, pp. 529–568. DOI: 10.1007/s12532-020-00183-6.

Zeng, B. and L. Zhao (2012). "An Exact Algorithm for Two-stage Robust Optimization with Mixed Integer Recourse Problems." In: *Optimization Online*. Published: 2012/01/10. URL: https://optimization-online.org/?p=11876.

— (2013). "Solving two-stage robust optimization problems using a column-and-constraint generation method." In: *Operations Research Letters* 41.5, pp. 457–461. DOI: 10.1016/j.orl.2013.05.003.