# idol: A C++ Framework for Optimization

## Reference Manual

For idol v1.0.0-alpha.

Henri Lefebvre

# Contents

# Getting Started

CHAPTER 1

# What is **idol**

# Part 1

# Mixed-Integer Optimization

CHAPTER 2

# Modeling a Mixed-Integer Problem

## Contents

## 1. Introduction

In idol, a mixed-integer optimization problem is specified in the general quadratic 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, $x \in \mathbb{R}^n$ denotes the decision vector. The objective comprises linear and quadratic components, defined by $c \in \mathbb{Q}^n$ and $D \in \mathbb{Q}^{n \times n}$, together with a constant term $c_0 \in \mathbb{Q}$. Each constraint $i$ consists of a linear part $a_{i\cdot} \in \mathbb{Q}^n$, a quadratic part $Q^i \in \mathbb{Q}^{n \times n}$, and a right-hand side $b_i \in \mathbb{Q}$. Variable bounds are given by $\ell_j, u_j \in \mathbb{Q} \cup \{-\infty, \infty\}$. The index set $J$ specifies the variables that are required to take integer values.

As is customary, variables are classified according to their type, i.e., continuous, integer, or binary, and according to their bounds. The terminology used in idol is summarized in Table 1. A constraint is said to be linear when $Q^i = 0$, and quadratic otherwise. Likewise, the objective function is considered quadratic whenever $D \neq 0$. A particularly important subclass of MIPs arises when both the constraints and the objective are linear; that is, when $Q^i = 0$ for all $i$ and $D = 0$. Such problems are known as mixed-integer linear programs (MILPs).

Remark 1. *You do not need to read every section in detail before formulating your first model. Most users can begin with the example in Section 2, which*

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 | $\ell = u$ |

*demonstrates how to define a basic MILP in idol. The subsequent sections develop the modeling framework in greater depth, including the treatment of variables, constraints, and the environment that manages them.*

*Note that solving a model, i.e., computing an optimal solution, is not covered in this chapter. This topic is addressed in the next chapter, where we introduce the notions of 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 given by the MILP

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

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 (in orange), and the optimization direction (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 objective value $-7.2$. The original integer-constrained problem also admits a unique solution, which is attained at $(x^*, y^*) = (2, 2)$, with objective value $-5$.

Modeling Problem (2) in idol is straightforward. If you are familiar with other optimization frameworks—such as JuMP in Julia or the Gurobi Python or C++ interface—the following code snippet should be largely self-explanatory.

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

using namespace idol;

int main(int t_argc, const char** t_argv) {

    Env env;

    // Create a new model.
    Model model(env);

```

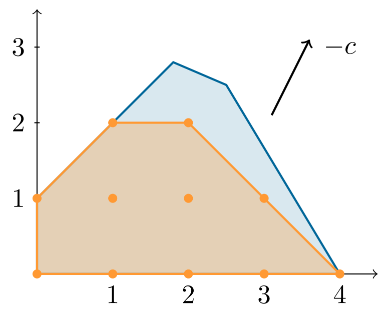

FIGURE 1. The toy example of a MILP considered in Section 2. The shaded blue area is the feasible region of the continuous relaxation. The dots in orange are the feasible points in Problem (2), while the shaded orange area denote the convex hull of these points.

```cpp
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 used for debugging. Finally, Lines 18–20 add constraints to the model to define the feasible region of the problem.

Note that at this stage we are only formulating Problem (2); no optimization is being performed yet. Solving the model is the focus of the next chapter. For completeness, however, the following example illustrates how the commercial solver Gurobi can be used to compute a solution to 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 is represented by the Env class. It 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 by this environment 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, adding a variable from one environment to a model created by a different environment will lead to an undefined behavior, often resulting in a segmentation fault and a program crash.

Creating an environment is straightforward.

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

Once initialized, the environment can be used to create models, variables, and constraints. All such objects are associated with env and are managed by it throughout their lifetime.

## 4. Models

Mathematical optimization problems are modeled using the Model class. A model consists of a collection of variables and constraints together with an objective function. It is created by invoking the constructor of Model and passing an environment as its 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 yet contain any variables or constraints. By default, all models are treated as minimization problems. Currently, idol offers only limited support for maximization problems. This is not a real restriction, however, since $\max_{x\in X} f(x) = -\min_{x\in X} -f(x)$ holds for all function $f$ and all set $X$.

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 decision to rely on external solvers is justified by two main considerations. First, idol is typically used in combination with such solvers anyway to efficiently solve optimization problems. Second, it is generally safer to leverage well-established solver implementations, which have been extensively tested over many years, to read and solve models 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.

```cpp
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.

```cpp
std::cout << "N. of vars: "         << model.vars().size()    << '\n';
std::cout << "N. of linear ctrs: "  << model.ctrs().size()    << '\n';
std::cout << "N. of quad. ctrs: "   << model.qctrs().size()   << '\n';
std::cout << "N. of SOS ctrs: "     << model.sosctrs().size() << '\n';
```

To obtain model-specific information about a variable, a constraint, or the objective function, idol provides methods of the form Model::get_X_Y(const X&), where X denotes the type of object—such as var, ctr, qctr, sosctr, or obj—and Y specifies the particular attribute to access, for example lb, type, or column. To give a specific example, the following code snippet counts the number of binary variables in the model.

```cpp
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;
    }

}
```

The complete set of information that can be accessed through a model is described in detail in the following sections.

In most practical situations, it is preferable to avoid copying a model and instead pass a reference to it to auxiliary functions. For this reason, the copy constructor of the Model class is declared private. If copying a model is indeed required, the Model::copy() method and the move constructor should be used. The following example illustrates this usage.

```cpp
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.

```cpp
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 constitute the decision-making elements of an optimization problem. They represent the quantities to be determined in order to optimize an objective function, subject to a set of constraints. In idol, variables are represented by the Var class.

**5.1. Creating Variables.** Creating variables can mainly be done in two ways. The first approach uses the Var constructor in combination with the Model::add() method, while the second relies on the Model::add_var(...) methods. We begin with the first approach, which employs the Var constructor. Although this method is somewhat less direct, it provides greater insight into how optimization objects are managed within idol.

We focus on the constructor given by

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

The constructor requires six arguments. The first specifies the optimization environment, which manages the variable's lifetime and versions. The next two arguments define the lower and upper bounds of the variable, which can be set to infinity using idol::Inf. The fourth argument indicates the type of the variable, for example idol::Continuous, idol::Integer, or idol::Binary. The fifth argument provides the linear coefficient of the variable in the objective function, and the sixth argument assigns a name to the variable.

For example, the following code creates a new variable within the environment.

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

This variable is continuous, non-negative, and has an objective coefficient of 2. It is named "x". Importantly, at this stage the variable does not belong to any model. What has been created is referred to as the "default version" of the variable. By default, if this variable is later added to a model, it will retain these attributes within that model. For example, the following code demonstrates how to create the variable and add it 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. Other constructors are also available in the Var class. For example, it is possible to provide a column associated with the variable so that it is automatically incorporated into the LP matrix. Columns can be constructed using the LinExpr<Ctr> class in a straightforward and intuitive manner. For more details, see Section 6, which covers expressions in idol. We provide one illustrative example here.

```
// 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 note the use of the TempVar class. This lightweight class is designed to represent a variable that has not yet been created within an environment. It stores all the attributes of the variable to be created, but cannot be used for any other purpose except for holding these attributes and instantiating an actual variable.

The second approach for creating variables is more straightforward, although it is internally equivalent to the method described above. This approach uses the Model::add_var methods of the Model class. The following code snippet illustrates this usage and should be easy to understand.

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

Note that it is not necessary to pass the environment explicitly, as the environment associated with the model is used automatically. In this single call, two operations are performed: first, a default version of the variable is created, and second, the variable is added to the model. Similarly, it is also possible to add a variable together with a specific column in the LP matrix.

In some cases, it is more convenient to create multiple variables at once. This can be accomplished using the Var::make_vector function or the Model::add_vars method. Both functions require an additional parameter specifying the dimension of the variable set. For example, the following illustrates how to create a set of variables indexed by a $2 \times 3$ grid.

```
// Create a (2,3) "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;
```

Note that we use the Dim class to specify the dimensions of the variable set. The Dim class is a template that takes an integer parameter, which indicates the number of indices for the new variable. In this example, we pass 2 to create a two-dimensional index. The size of each dimension is then specified by providing the appropriate arguments to the constructor of Dim, namely 2 and 3.

Alternatively, the same result can be achieved using methods of the Model class. The following snippet illustrates this approach.

```
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 using the Model::remove(const Var&) method. This operation removes the variable from the model and updates all linear and quadratic constraints in which the variable appears. Attempting to remove a variable that does not belong to the model will result in an exception. To check whether a variable is part of a model, the Model::has(const Var&) method can be used; it returns true if and only if the variable is included in the model.

Note that variables involved in SOS-type constraints cannot be removed directly. This restriction is not limiting in practice, as SOS constraints themselves can be removed and added again if needed.

**5.3. Accessing Variables.** Variables possess two immutable attributes: a *name*, assigned at the time of creation, and a unique *id* within the environment. Other attributes are model-specific and can be accessed through the model's methods Model::get_var_Y, where Y denotes the name of the attribute. The following list summarizes the methods available for retrieving 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.

The following example demonstrates how to print all free variables in a model.

```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 variable indices. Although indices may change over time, e.g., if variables are removed from a model, they can still be used to access variables via the Model::get_var_by_index method. The following code snippet illustrates an alternative way to iterate over all 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 attributes of a variable can be modified directly through the model's methods Model::set_var_Y, where Y denotes the attribute to be changed. The following list summarizes the available methods for modifying variable attributes 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 conclude with an example demonstrating how to copy a model and create 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 3

# Solving Problems with Optimizers 🚧

## Contents

## 1. Optimizers and Optimizer Factories 🚧

## 2. Example: The Binary Knapsack Problem 🚧

## 3. External Solvers 🚧

### 3.1. Cplex.

### 3.2. GLPK 🚧.

### 3.3. Gurobi 🚧.

### 3.4. HiGHS 🚧.

### 3.5. JuMP 🚧.

### 3.6. Mosek 🚧.

## 4. Creating Your Own Optimizer 🚧

CHAPTER 4

# 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 5

# Implementing a Custom Branch-and-Bound Algorithm 🚧

**Contents**

## 1. The Branch-and-Bound Algorithm 🚧

## 2. Implementation in idol 🚧

## 3. Branching Rules 🚧

## 4. Node Selection Rules 🚧

## 5. Node Types 🚧

## 6. idol-Specific Callbacks 🚧

## 7. Example: A Knapsack-specific Branch-and-Bound 🚧

CHAPTER 6

# Column Generation and Branch-and-Price 🚧

## Contents

Column generation and branch-and-price are advanced techniques for solving large-scale integer programming problems. This chapter introduces the fundamental concepts and demonstrates how to implement these techniques using idol. For a detailed treatment of the theoretical foundations, we refer the reader to the books by Desrosiers and Desaulniers (2024) and Uchoa et al. (2024).

The chapter is organized as follows. In Section 1, we present the general theory of Dantzig-Wolfe decomposition for linear programs. Section 2 then describes how to solve integer programs by embedding the Dantzig-Wolfe decomposition within a branch-and-bound framework, thereby obtaining a branch-and-price algorithm.

## 1. The Dantzig-Wolfe Decomposition 🚧

**1.1. Introduction.** The Dantzig-Wolfe decomposition is a decomposition technique that exploits the block structure of large-scale linear programs. It was first introduced by Dantzig and Wolfe (1960). Formally, we consider problems of the form

$$\min_{x^1,\dots,x^K} \quad \sum_{k=1}^K {c^k}^\top x^k \tag{3a}$$

$$\text{s.t.} \quad \sum_{k=1}^K A^k x^k \ge b, \tag{3b}$$

$$B^k x^k \ge d^k, \quad \text{for all } k = \dots, K. \tag{3c}$$

Here, Constraints (3b) are considered to be complicating constraints in the sense that, without them, the problem decomposes into $K$ independent subproblems of the form

$$\min_{x^k} \left\{ {c^k}^\top x^k : B^k x^k \ge d^k \right\},$$

for all $k = 1, \dots, K$. For the sake of simplicity, we first assume that the polyhedron $P^k := \{x : B^k x \ge d^k\}$ is actually a polytope, i.e., it is bounded. The goal of the

Dantzig-Wolfe decomposition is to exploit this block structure to solve the original problem more efficiently.

We start by reformulating Problem (3). In particular, Constraints (3c) are equivalently expressed by representing each $x^k$ as a convex combination of points $\hat{x}^k$ satisfying $B^k \hat{x}^k \geq d^k$ for each $k = 1, \ldots, K$. That is, we can write

$$x^k = \sum_{r \in \mathcal{R}^k} \lambda_r^k \, \hat{x}_r^k, \qquad \sum_{r \in \mathcal{R}^k} \lambda_r^k = 1, \qquad \lambda_r^k \geq 0 \quad \forall r \in \mathcal{R}^k,$$

where $\mathcal{R}^k$ denotes the set of extreme points of the $k$-th polytope $P^k$, and $\lambda_r^k$ are the associated convex combination coefficients.

Substituting this representation into the objective function and the complicating constraints (3b) yields the *Dantzig-Wolfe reformulation*:

$$\min_{\lambda} \quad \sum_{k=1}^{K} \sum_{r \in \mathcal{R}^k} {c^k}^\top \hat{x}_r^k \, \lambda_r^k \tag{4a}$$

$$\text{s.t.} \quad \sum_{k=1}^{K} \sum_{r \in \mathcal{R}^k} A^k \hat{x}_r^k \, \lambda_r^k \geq b, \tag{4b}$$

$$\sum_{r \in \mathcal{R}^k} \lambda_r^k = 1, \quad \forall k = 1, \ldots, K, \tag{4c}$$

$$\lambda_r^k \geq 0, \quad \forall k = 1, \ldots, K, \ r \in \mathcal{R}^k. \tag{4d}$$

In practice, the sets $\mathcal{R}^k$ are typically very large or even exponential in size. Column generation is therefore used to iteratively add only the columns (variables) corresponding to promising extreme points, allowing the master problem to remain tractable while still converging to the optimal solution of the original problem.

### 1.2. Column Generation.

### 1.3. Stabilization via Dual-Smoothing.

## 2. The Branch-and-Price Algorithm 🚧

### 2.1. The Algorithm.

### 2.2. Hard vs. Soft Branching.

## 3. Implementation

## 4. Example: The Generalized Assignment Problem 🚧

## 5. Strong Branching 🚧

CHAPTER 7

# Penalty Alternating Direction Method 🚧

**Part 2**

# Bilevel Optimization

CHAPTER 8

# Modeling a Bilevel Problem 🚧

Contents

## 1. Introduction 🚧

## 2. The Bilevel::Description Class 🚧

## 3. Reading and Writing Instance Files from BOBILib 🚧

CHAPTER 9

# Problems with continuous follower 🚧

## Contents

$$\begin{aligned}
\min_{x,y} \quad & c^\top x + d^\top y && \text{(5a)}\\
\text{s.t.} \quad & Ax + By \geq a, && \text{(5b)}\\
& y \in S(x). && \text{(5c)}
\end{aligned}$$

## 1. The Strong-Duality-Based Single-Level Reformulation 🚧

## 2. The KKT-Based Single-Level Reformulation 🚧

## 3. Linearization Techniques for the KKT-Based Single-level Reformulation 🚧

$$\begin{aligned}
\min_{y} \quad & f^\top y && && \text{(6a)}\\
\text{s.t.} \quad & C^= x + D^= y = b^=, && (\lambda^= \in \mathbb{R}^{m_=}) && \text{(6b)}\\
& C^\leq x + D^\leq y \leq b^\leq, && (\lambda^\leq \in \mathbb{R}^{m_\leq}_{\leq 0}) && \text{(6c)}\\
& C^\geq x + D^\geq y \geq b^\geq, && (\lambda^\geq \in \mathbb{R}^{m_\geq}_{\geq 0}) && \text{(6d)}\\
& y \leq y^\leq, && (\pi^\leq \in \mathbb{R}^{n}_{\leq 0}) && \text{(6e)}\\
& y \geq y^\geq && (\pi^\geq \in \mathbb{R}^{n}_{\geq 0}). && \text{(6f)}
\end{aligned}$$

$$\begin{aligned}
\max_{\lambda^=,\lambda^\geq,\lambda^\leq,\pi^\leq,\pi^\geq} \quad & (b^= - C^= x)^\top \lambda^= + (b^\leq - C^\leq x)^\top \lambda^\leq + (b^\geq - C^\geq x)^\top \lambda^\geq && \text{(7a)}\\
& + \sum_{j: y_j^\leq < \infty} (y^\leq)^\top \pi^\leq + \sum_{j: y_j^\geq > -\infty} (y^\geq)^\top \pi^\geq && \text{(7b)}\\
\text{s.t.} \quad & (D^=)^\top \lambda^= + (D^\leq)^\top \lambda^\leq + (D^\geq)^\top \lambda^\geq + \pi^\leq + \pi^\geq = d, && \text{(7c)}\\
& \lambda^\leq \leq 0, \lambda^\geq \geq 0, \pi^\leq \leq 0, \pi^\geq \geq 0. && \text{(7d)}
\end{aligned}$$

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.**

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

**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. Example: A Toy Linear 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 10

# 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 11

# Pessimistic Bilevel Optimization 🚧

## Contents

## 1. Introduction 🚧

## 2. Reformulation as an Optimistic Bilevel Problem 🚧

# Part 3

# Robust Optimization

CHAPTER 12

# 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 13

# Deterministic Reformulations 🚧

## Contents

## 1. Duality-Based Reformulations 🚧

## 2. Bilevel Reformulations 🚧

CHAPTER 14

# 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. 2024 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

$$\min_{x\in X} c^\top x + \max_{u\in U} \min_{y\in Y(x,u)} d^\top y \tag{10}$$

where $X$ denotes the set of feasible first-stage decisions, $U$ denotes the uncertainty 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 (10) 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 (10) having replaced the uncertainty set $U$ by a discrete subset of scenarios $\{u^1, \dots, u^K\}$. This problem can be formulated as

$$\begin{aligned} \min_{x\in X, x_0, y^1,\dots,y^K \in \tilde{Y}} \quad & c^\top x + x_0 && \text{(11a)} \\ \text{s.t.} \quad & x_0 \geq d^\top y^k, \quad \text{for all } k = 1, \dots, K, && \text{(11b)} \\ & Cx + Dy^k + E(x)u^k \geq b, \quad \text{for all } k = 1, \dots, K. && \text{(11c)} \end{aligned}$$

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 (11) 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 (11) is infeasible, then (10) is infeasible and the algorithm stops.

Now, assume that the restricted master problem (11) 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 (10), 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 (11) $c^\top x + x_0$ 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

$$v_{\text{feas}} := \max_{u \in U} \min_{y,z} \left\{ e^\top z \colon y \in \tilde{Y},\ z \geq 0,\ Cx + Dy + E(x)u + z \geq b \right\}. \tag{12}$$

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 (11) 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 (12) 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

$$v_{\text{opt}} := \max_{u \in U} \min_{y} \left\{ d^\top y \colon y \in \tilde{Y},\ Cx + Dy + E(x)u \geq b \right\}. \tag{13}$$

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 (13) is always feasible. Note that $c^\top x + v_{\text{opt}}$ yields an upper bound on the optimal objective function value of Problem (10). 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 (11). A scenario $u$ which disproves the optimality of $x$ is always added to the master problem (11) 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

$$\begin{aligned} -\min_{u \in U, y} \quad & -d^\top y \\ \text{s.t.} \quad & y \in \arg\min_{\bar{y}} \left\{ d^\top \bar{y} \colon \bar{y} \in \tilde{Y},\ Cx + D\bar{y} + E(x)\bar{u} \geq b \right\}. \end{aligned} \tag{14}$$

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

```
1: ...
2: while  do
3:     ...
4:     if true then
5:         ...
6:     end if
7:     ...
8: 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

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

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 (10). 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

```
1: ...
2: while  do
3:     ...
4:     if true then
5:         ...
6:     end if
7:     ...
8: 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 12. 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 (11). 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 (13)—which, internally is modeled as (14)—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 $(y, \lambda)$ 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 (13) by its KKT conditions, leading to the nonlinear problem

$$\begin{aligned} \max_{u,y,\lambda} \quad & d^\top y && \text{(15a)} \\ \text{s.t.} \quad & u \in U, && \text{(15b)} \\ & Cx + Dy + E(x)u \geq b, && \text{(15c)} \\ & D^\top \lambda = d, \quad \lambda \geq 0, && \text{(15d)} \\ & (Cx + Dy + E(x)u - b)^\top \lambda = 0. && \text{(15e)} \end{aligned}$$

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 (15e) by introducing auxiliary binary variables. However, this approach requires to compute bounds on the dual variables $\lambda$ 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 (15e) 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 (15e) 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 10 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 (15e) that does not require computing big-M values is through SOS1 constraints. Again, we refer to Chapter 10 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 (13) 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 (13) leads to the nonlinear model

$$\max_{u \in U, \lambda} \left\{ (b - Cx - E(x)u)^\top \lambda \colon D^\top \lambda = d, \ \lambda \geq 0 \right\} \tag{16}$$

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 uppon the duality-based separation, consider problem (16) 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 $\mu$. 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

$$\min_{x,y,z} \quad \sum_{i \in V_1} f_i x_i + \sum_{i \in V_1} \sum_{j \in V_2} c_{ij} d_j y_{ij} + \sum_{j \in V_2} p_j d_j z_j \tag{19a}$$

$$\text{s.t.} \quad \sum_{i \in V_1} y_{ij} + z_j = 1, \quad \text{for all } j \in V_2, \tag{19b}$$

$$y_{ij} \leq x_i, \quad \text{for all } i \in V_1, \text{for all } j \in V_2, \tag{19c}$$

$$y_{ij} \geq 0, z_j \geq 0, \quad \text{for all } i \in V_1, j \in V_2, \tag{19d}$$

$$x_i \in \{0, 1\}, \quad \text{for all } i \in V_1. \tag{19e}$$

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

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

where, for all facility $i \in V_1$, $u_i = 1$ if and only if faciltiy $i$ is disrupted. The parameter $\Gamma$ 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

$$Y(x,u) := \left\{ y \in \mathbb{R}^{|V_1|\times|V_2|} \colon \text{(19b)–(19d) and } y_{ij} \le 1 - u_i, \quad \text{for all } i \in V_1 \right\}.$$

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 12, we first need to create the deterministic model (19). 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 (19c). 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 "$-x_iu_i$" 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.,

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

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 (19). 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

$$\max_{\alpha,\beta,\gamma,\delta} \quad \sum_{j\in V_2} \alpha_j + \sum_{i\in V_1}\sum_{j\in V_2} x_i(1-u_i)\beta_{ij} \tag{20a}$$
$$\text{s.t.} \quad \alpha_j + \beta_{ij} + \gamma_{ij} = c_{ij}d_j, \quad \text{for all } i\in V_1, j\in V_2, \tag{20b}$$
$$\alpha_j + \delta_j = p_j d_j, \quad \text{for all } j \in V_2, \tag{20c}$$
$$\alpha_j \in \mathbb{R}, \beta_{ij}\le 0, \gamma_{ij}\ge 0, \delta_j \ge 0, \quad \text{for all } i\in V_1, j\in V_2. \tag{20d}$$

Hence, the separation problem reads

$$\max_{u,\alpha,\beta,\gamma,\delta} \quad \sum_{j\in V_2} \alpha_j + \sum_{i\in V_1}\sum_{j\in V_2} x_i(1-u_i)\beta_{ij}$$
$$\text{s.t.} \quad u\in U, (20b)\text{–}(20d).$$

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

```cpp
    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 10 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

$$M_{ij}v_{ij} \le w_{ij} \le 0, \quad \beta_{ij} \le w_{ij} \le \beta_{ij} - M_{ij}(1-v_{ij}), \quad v_{ij}\in\{0,1\}, \tag{21}$$

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 10, 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 $\beta$ since $\beta$ is the only variable involved in a product. For that reason, only the get_ctr_dual_lb method is implemented ($\beta$ is known to be non-negative).

```cpp
    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.

```cpp
    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.

```cpp
    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

$$\begin{aligned}
\text{primal feasibility} &= \begin{cases} \sum_{i\in V_1} y_{ij} + z_j = 1, & \text{for all } j \in V_2,\\ y_{ij} \le x_i(1-u_i), & \text{for all } i \in V_1, \text{for all } j \in V_2,\\ y_{ij} \ge 0, z_j \ge 0, & \text{for all } i \in V_1, j \in V_2, \end{cases}\\
\text{dual feasibility} &= \begin{cases} \alpha_j + \beta_{ij} + \gamma_{ij} = c_{ij} d_j, & \text{for all } i \in V_1, j \in V_2,\\ \alpha_j + \delta_j = p_j d_j, & \text{for all } j \in V_2, \end{cases}\\
\text{stationarity} &= \begin{cases} \alpha_j \in \mathbb{R}, \beta_{ij} \le 0, \gamma_{ij} \ge 0, \delta_j \ge 0, & \text{for all } i \in V_1, j \in V_2. \end{cases}\\
\text{complementarity} &= \begin{cases} \beta_{ij}(y_{ij} - x_i(1-u_i)) = 0, & \text{for all } i \in V_1, \text{for all } j \in V_2,\\ \gamma_{ij} y_{ij} = 0, & \text{for all } i \in V_1, \text{for all } j \in V_2,\\ \delta_j z_j = 0, & \text{for all } i \in V_1, \text{for all } j \in V_2. \end{cases}
\end{aligned} \tag{22}$$

Hence, the separation problem can be formulated as

$$\max_{u,y,\alpha,\beta,\gamma,\delta} \left\{ d^\top y \colon u \in U, (22) \right\}.$$

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

```cpp
    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 $\beta_{ij} \geq \min\{0, d_j(c_{ij} - p_j)\}$. 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 15

# Objective Uncertainty 🚧

CHAPTER 16

# 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.

Dantzig, G. B. and P. Wolfe (1960). “Decomposition Principle for Linear Programs.” In: *Operations Research* 8.1, pp. 101–111. DOI: 10.1287/opre.8.1.101.

Desrosiers, J. and G. Desaulniers (2024). *Branch-and-price.* URL: https://www.researchgate.net/profile/Jacques-Desrosiers/publication/381918232_Branch-and-Price/links/6739dbc34a70511f07226464/Branch-and-Price.pdf.

Khademi, A., A. Marandi, and M. Soleimani-damaneh (2024). “A new dual-based cutting plane algorithm for nonlinear adjustable robust optimization.” In: *Journal of Global Optimization* 89.3, pp. 559–595. DOI: 10.1007/s10898-023-01360-2.

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). “Remarks on: A Lagrangian dual method for two-stage robust optimization with binary uncertainties.” In: *Optimization and Engineering.* DOI: 10.1007/s11081-025-10015-y.

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.

Uchoa, E., A. Pessoa, and L. Moreno (2024). *Optimizing with Column Generation: Advanced Branch-Cut-and-Price Algorithms.* Tech. rep. L-2024-3. Cadernos do LOGIS-UFF, Universidade Federal Fluminense, Engenharia de Produção. URL: https://optimizingwithcolumngeneration.github.io.

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.