Tutorial on CPLEX
Linear Programming

Combinatorial Problem Solving (CPS)

Enric Rodríguez-Carbonell

April 28, 2021

LP with CPLEX

- Among other things, CPLEX allows one to deal with:
  - Real linear progs (all vars are in $\mathbb{R}$)

$$\begin{aligned} \min \quad & c^T x \\ & A_1 x \leq b_1 \\ & A_2 x = b_2 \\ & A_3 x \geq b_3 \\ & x \in \mathbb{R}^n \end{aligned}$$

  - Mixed integer linear progs (some vars are in $\mathbb{Z}$)

$$\begin{aligned} \min \quad & c^T x \\ & A_1 x \leq b_1 \\ & A_2 x = b_2 \\ & A_3 x \geq b_3 \\ & \forall i \in I : x_i \in \mathbb{Z} \\ & \forall i \notin I : x_i \in \mathbb{R} \end{aligned}$$

CPLEX Toolkit

- CPLEX allows one to work in several ways. CPLEX is...
  - An IDE that uses the OPL modeling language
  - An interactive command-line optimizer that reads MPS/LP input
  - A callable library in several languages
    - Java
    - C
    - C++ (Concert Technology)
    - ...

Concert Technology

- Two kinds of objects:

  - Modeling objects for defining the optimization problem (constraints, objective function, etc.)

    They are grouped into an `IloModel` object representing the complete optimization problem (recall: here, model = problem).

  - Solving objects for solving problems represented by modeling objects.

    An `IloCplex` object reads a model, extracts its data, solves the problem and answers queries on solution.

Creating the Environment: `IloEnv`

- The class `IloEnv` constructs a CPLEX environment.
- The environment is the first object created in an application.
- To create an environment named `env`, you do this:

```
IloEnv env;
```

- The environment object needs to be available to the constructor of all other Concert Technology classes
- `IloEnv` is a handle class: variable `env` is a pointer to an implementation object, which is created at the same time
- Before terminating destroy the implementation object with

```
env.end();
```

  for just ONE of its `IloEnv` handles

Creating a Model: `IloModel`

- After creating the environment, a Concert application is ready to create one or more optimization models.
- Objects of class `IloModel` define a complete model (which can be solved by passing it to an `IloCplex` object).
- To construct a modeling object named `model`, within an existing environment named `env`, call:

```
IloModel model(env);
```

- One can get the environment of a given optimization model by calling:

```
IloEnv env = model.getEnv();
```

Creating a Model: `IloModel`

- After an `IloModel` object has been constructed, it can be populated with objects of classes:
  - `IloNumVar`, representing modeling variables;
  - `IloRange`, which define constraints of the form $l \leq E \leq u$, where E is a linear expression;
  - `IloObjective`, representing an objective function.
- Any object `obj` can be added to the model by calling:

```
model.add(obj);
```

- No need to explicitly add the variable objects to a model, as they are implicitly added when they are used in range constraints (instances of `IloRange`) or in the objective.
- At most one objective function can added to a model.

Creating a Model: `IloModel`

- Modeling variables are constructed as objects of class `IloNumVar`, e.g.:

```
IloNumVar x(env, 0, 40, ILOFLOAT);
```

  This definition creates the modeling variable `x` with lower bound 0, upper bound 40 and type `ILOFLOAT`

- Variable types:
  - `ILOFLOAT`: continuous variable
  - `ILOINT`: integer variable
  - `ILOBOOL`: Boolean variable

- Note that even for variables of type `ILOBOOL`,
  the lower bound (0) and the upper bound (1) have to be specified.

- One may have arrays of variables: `IloNumVarArray`

Creating a Model: `IloModel`

- After all the modeling variables have been constructed, they can be used to define the objective function (objects of class `IloObjective`) and constraints (objects of class `IloRange`).

- To create `obj` of type `IloObjective` representing an objective function (and direction of optimization):

```
IloObjective obj = IloMinimize(env, x+2*y);
```

or

```
IloObjective obj = IloMaximize(env, x+2*y);
```

- Creating constraints and adding them to the model:

```
model.add(-x + 2*y + z <= 20);
```

`-x + 2*y + z <= 20` creates implicitly an object of class `IloRange` that is immediately added to the model

Creating a Model: `IloModel`

- Actually in

  ```
  model.add(-x + 2*y + z <= 20);
  ```

  an object of class `IloExpr` (a linear expression) is also implicitly created before the object of class `IloRange` is created

- Sometimes it is convenient to create objects of class `IloExpr` explicitly.

  E.g., when expressions cannot be spelled out in source code but have to be built up dynamically. Operators like += provide an efficient way to do this.

- Example:

  ```
  IloEnv         env;
  IloModel model(env);
  IloNumVarArray V(env, n, 0, 1, ILOBOOL);
  IloExpr expr(env);
  for (int i = 0; i < n; ++i) expr += V[i];
  model.add(expr == 1);
  expr.end();
  ```

Creating a Model: `IloModel`

```
IloEnv          env;
IloModel model(env);
IloNumVarArray V(env, n, 0, 1, ILOBOOL);
IloExpr expr(env);
for (int i = 0; i < n; ++i) expr += V[i];
model.add(expr == 1);
expr.end(); // Notice this!
```

- `IloExpr` objects are handles.
  So if an `IloExpr` object has been created explicitly,
  then method `end()` must be called when it is no longer needed.

- Example of explicitly defining an `IloRange` object

```
IloRange c(env, -IloInfinity, -x1 + x2 + x3, 0);
```

- One may have arrays of constraints: `IloRangeArray`

Solving the Model: `IloCplex`

- The class `IloCplex` solves a model.
- After the optimization problem has been stored in an `IloModel` object (say, `model`), it is time to create an `IloCplex` object (say, `cplex`) for solving the problem:

```
IloCplex cplex(model);
```

- To solve the model, call:

```
cplex.solve ();
```

- This method returns an `IloBool` value, where:
  - `IloTrue` indicates that CPLEX successfully found a feasible (yet not necessarily optimal) solution
  - `IloFalse` indicates that no solution was found

Solving the Model: `IloCplex`

- More precise information about the outcome of the last call to the method `solve` can be obtained by calling:

```
cplex.getStatus ();
```

- Returned value tells what CPLEX found out: whether
    - it found the optimal solution or only a feasible one; or
    - it proved the model to be unbounded or infeasible; or
    - nothing at all has been proved at this point.

- More info is available with method `getCplexStatus`.

Querying Results

- Query methods access information about the solution.
- Numbers in solution, etc. are of type `IloNum` (= `double`)

- To query the solution value for a variable:

```
IloNum v = cplex.getValue(x);
```

- Warning! Sometimes for integer variables the value is not integer but just “almost” integer (e.g. 1e-9 instead of 0).

  Explicit rounding necessary
  (use functions `round` of `<math.h>` or `IloRound`).

- To query the solution value for an array of variables:

```
IloNumVarArray x(env);
...
IloNumArray v(env);
cplex.getValues(v, x);
```

Querying Results

- To get the values of the slacks of an array of constraints:

```
IloRangeArray c(env);
...
IloNumArray v(env);
cplex.getSlacks(v, c);
```

- To get the values of the dual variables (simplex multipliers) of an array of constraints:

```
IloRangeArray c(env);
...
IloNumArray v(env);
cplex.getDuals(v, c);
```

Querying Results

- To get the values of the reduced costs of an array of variables:

```
IloNumVarArray x(env);
...
IloNumArray v(env);
cplex.getReducedCosts(v, x);
```

- To avoid logging messages by CPLEX on screen:

```
cplex.setOut  (env.getNullStream());
cplex.setError(env.getNullStream());
```

Querying Results

- Output operator << is defined for type `IloAlgorithm::Status` returned by `getStatus`, as well as for `IloNum`, `IloNumVar`, ...

  << is also defined for any array of elements
  if the output operator is defined for the elements.

- Default names are of the form `IloNumVar`(n)`[`ℓ`..`u`]` for variables, and similarly for constraints, e.g.,

```
IloNumVar(1)[0..9]  + IloNumVar(3)[0..inf]  <= 20
```

- One can set names to variables and constraints:

```
    x.setName("x");
    c.setName("c");
```

Writing/Reading Models

- CPLEX supports reading models from files and writing models to files in several languages (e.g., LP format, MPS format)

- To write the model to a file (say, `model.lp`):

```
cplex.exportModel ("model.lp");
```

- `IloCplex` decides which file format to write based on the extension of the file name (e.g., `.lp` is for LP format)

- This may be useful, for example, for debugging

Languages for Linear Programs

- *MPS*
  - Very old format ($\approx$ age of punched cards!) by IBM
  - Has become industry standard over the years
  - Column-oriented
  - Not really human-readable nor comfortable for writing
  - All LP solvers support this language
- *LP*
  - CPLEX specific file format
  - Row-oriented
  - Very readable, close to mathematical formulation
  - Supported by CPLEX, GUROBI, GLPK, LP_SOLVE, ..
    (which can translate from one format to the other!)

Example: Product Mix Problem

- A company can produce 6 different products $P_1, \ldots, P_6$
- Production requires labour, energy and machines, which are all limited
- The company wants to maximize revenue
- The next table describes the requirements of producing one unit of each product, the corresponding revenue and the availability of resources:

	P_1	P_2	P_3	P_4	P_5	P_6	Limit
Revenue	5	6	7	5	6	7	
Machine	2	3	2	1	1	3	1050
Labour	2	1	3	1	3	2	1050
Energy	1	2	1	4	1	2	1080

Example: Product Mix Problem

MODEL:

$x_i =$ quantity of product P_i to be produced.

$$
\begin{array}{llllllll}
\text{max Revenue}: & 5x_1 & +6x_2 & +7x_3 & +5x_4 & +6x_5 & +7x_6 & \\
\text{Machine}: & 2x_1 & +3x_2 & +2x_3 & +x_4 & +x_5 & +3x_6 & \leq 1050 \\
\text{Labour}: & 2x_1 & +x_2 & +3x_3 & +x_4 & +3x_5 & +2x_6 & \leq 1050 \\
\text{Energy}: & 1x_1 & +2x_2 & +x_3 & +4x_4 & +x_5 & +2x_6 & \leq 1080 \\
 & x_1, & x_2, & x_3, & x_4, & x_5, & x_6 & \geq 0
\end{array}
$$

LP Format

```
\ Product-mix problem (LP format)

max
revenue: 5 x_1  + 6 x_2  + 7 x_3  + 5 x_4  + 6 x_5  + 7 x_6

subject to

machine: 2 x_1  + 3 x_2  + 2 x_3  +  x_4   +  x_5   + 3 x_6   <= 1050
labour:  2 x_1  +  x_2   + 3 x_3  +  x_4   + 3 x_5  + 2 x_6   <= 1050
energy:  1 x_1  + 2 x_2  +  x_3   + 4 x_4  +  x_5   + 2 x_6   <= 1080

end
```

MPS Format

```
* Product-mix problem (Fixed MPS format)
*
* Column indices
*00000000111111111122222222223333333333444444444455555555556666666666
*23456789012345678901234567890123456789012345678901234567890123456789
*
* mrevenue stands for -revenue
*
NAME          PRODMIX
ROWS
 N  mrevenue
 L  machine
 L  labour
 L  energy
COLUMNS
    x_1       mrevenue            -5   machine             2
    x_1       labour               2   energy              1
    x_2       mrevenue            -6   machine             3
    x_2       labour               1   energy              2
    x_3       mrevenue            -7   machine             2
    x_3       labour               3   energy              1
    x_4       mrevenue            -5   machine             1
    x_4       labour               1   energy              4
    x_5       mrevenue            -6   machine             1
    x_5       labour               3   energy              1
    x_6       mrevenue            -7   machine             3
    x_6       labour               2   energy              2
RHS
    RHS1      machine           1050   labour           1050
    RHS1      energy            1080
ENDATA
```

LP Format

- Intended for representing LP's of the form

$$
\begin{array}{ll}
\min / \max \ c^T x & \\
a_i^T x \bowtie_i b_i & (1 \leq i \leq m, \ \bowtie_i \in \{\leq, =, \geq\}) \\
\ell \leq x \leq u & (-\infty \leq \ell_k, u_k \leq +\infty)
\end{array}
$$

- Comments: anything from a backslash \ to end of line
- In general blank spaces are ignored
  (except for separating keywords)
- Names are sequences of alphanumeric chars and symbols (,) _ etc.
  Careful with e, E: troubles with exponential notation!

LP Format

1. Objective function section
   (a) One of the keywords: `min`, `max`
   (b) Label with colon: e.g. `cost:` (optional)
   (c) Expression: e.g. `-2 x1 + 2 x2`
2. Constraints section
   (a) Keyword `subject to` (or equivalently: `s.t.`, `st`, `such that`)
   (b) List of constraints, each in a different line
      i. Label with colon: e.g. `limit:` (optional)
      ii. Expression: e.g. `3 x1 + 2 x2 <= 4`
      Senses: `<=`, `=<` for $\leq$; `>=`, `=>` for $\geq$; `=` for $=$

LP Format

3. Bounds section (optional)

   (a) Keyword `Bounds`

   (b) List of bounds, each in a different line

`l <= x <= u`:	sets lower and upper bounds
`l <= x` :	sets lower bound
`x >= l` :	sets lower bound
`x <= u` :	sets upper bound
`x = f` :	sets a fixed value
`x free` :	specifies a free variable

   (c) Infinite bounds $-\infty$, $+\infty$ are represented `-inf`, `+inf`

   (d) Default bounds: lower bound 0, upper bound $+\infty$

4. Generals section: Keyword `Generals` + list of integer variables (optional)
5. Binary section: Keyword `Binary` + list of binary variables (optional)
6. End section: File should end with keyword `End`

Writing/Reading Models

- `IloCplex` supports reading files with `importModel`

  A call to `importModel` causes CPLEX to read a problem from a file and add all data in it as new objects.

```
void IloCplex::importModel (
   IloModel&       m,
   const char*     filename,
   IloObjective&   obj,
   IloNumVarArray  vars,
   IloRangeArray   rngs) const;
```

Example 1

- Let us see a program for solving:

$$
\begin{aligned}
\max \quad & x_0 + 2x_1 + 3x_2 & \\
-x_0 + x_1 + x_2 \leq & \quad 20 \\
x_0 - 3x_1 + x_2 \leq & \quad 30 \\
0 \leq x_0 \leq & \quad 40 \\
0 \leq x_1 \leq & \quad \infty \\
0 \leq x_2 \leq & \quad \infty \\
x_i \in \mathbb{R} &
\end{aligned}
$$

Example 1

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN
int main () {
  IloEnv              env;
  IloModel       model(env);
  IloNumVarArray    x(env);
  x.add(IloNumVar(env, 0, 40));
  x.add(IloNumVar(env));//default: between 0 and +∞
  x.add(IloNumVar(env));
  model.add( - x[0] +     x[1] + x[2] <= 20);
  model.add(   x[0] - 3 * x[1] + x[2] <= 30);
  model.add(IloMaximize(env, x[0]+2*x[1]+3*x[2]));
  IloCplex cplex(model);
  cplex.solve();
  cout << "Max=" << cplex.getObjValue() << endl;
  env.end();
}
```

Example 2

- Let us see a program for solving:

$$
\begin{array}{lrl}
\max & x_0 + 2x_1 + 3x_2 + x_3 & \\
 & -x_0 + x_1 + x_2 + 10x_3 \leq & 20 \\
 & x_0 - 3x_1 + x_2 \leq & 30 \\
 & x_1 - 3.5x_3 = & 0 \\
 & 0 \leq x_0 \leq & 40 \\
 & 0 \leq x_1 \leq & \infty \\
 & 0 \leq x_2 \leq & \infty \\
 & 2 \leq x_3 \leq & 3 \\
 & x_0, x_1, x_2 \in \mathbb{R} & \\
 & x_3 \in \mathbb{Z} &
\end{array}
$$

Example 2

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN
int main () {
  IloEnv           env;
  IloModel    model(env);
  IloNumVarArray x(env);
  x.add(IloNumVar(env, 0, 40));
  x.add(IloNumVar(env));
  x.add(IloNumVar(env));
  x.add(IloNumVar(env, 2, 3, ILOINT));
  model.add( - x[0] + x[1] + x[2] + 10 * x[3] <= 20);
  model.add( x[0] - 3 * x[1] + x[2] <= 30);
  model.add( x[1] - 3.5* x[3] ==  0);
  model.add(IloMaximize(env, x[0]+2*x[1]+3*x[2]+x[3]));
  IloCplex cplex(model); cplex.solve();
  cout << "Max=" << cplex.getObjValue() << endl;
  env.end();
}
```

More information

- You can find collection of examples in lab's machines at:

  `/opt/ibm/ILOG/CPLEX_Studio124/cplex/examples/src/cpp`
  `/opt/ibm/ILOG/CPLEX_Studio124/cplex/examples/data`

- You can find a template for `Makefile` and the examples shown here at:

  `www.cs.upc.edu/~erodri/webpage/cps/lab/lp/tutorial-cplex-code/tutorial-cplex-code.tgz`