CONCISE CPLEX

Simon A. Spacey

Department of Computing
Imperial College
London, UK

ABSTRACT

This paper is a concise guide to CPLEX, the leading solver for linear and convex quadratic optimisation problems. The paper is self contained and includes information for first time CPLEX users as well as code snippets and lemmas that may be of referential value to experienced users.

The paper starts with a brief explanation of how to run CPLEX on departmental servers at Imperial and on stand-alone machines in section 1, how to create and solve simple Linear Programs in section 2 and how to obtain detailed solution results in section 3. The paper then moves on to discuss several CPLEX issues and quirks that may confuse first time users including: anomalous objective values caused by big-M scaling, the implications of long MILP solution times and removing memory limitations for problems with large MILP solution trees. The paper concludes with logical equivalence proofs in section 9 that can be used as a starting point for complex problem translation and references are provided for additional reading.

1. STARTING CPLEX

CPLEX [1] is installed in a single directory and can be moved from one machine to another with simple file copying. However, to execute CPLEX you need a license file, a defined hostname consistent with the license file and a license server to validate the license file. You can run CPLEX from its install directory on an Imperial server with the commands:

```
./ilm/ilmd &
./bin/x86-64_debian4.0_4.1/cplex
```

The first command starts the CPLEX license server and may not be necessary if the license server is already running as a shared process.

To execute a local copy of CPLEX from outside Imperial you will need to obtain a license file, set your local hostname to be consistent with the license file and use a license tunnel to Imperial so that CPLEX can validate the license and keep track of the current licenses in use. This can be automated using a script such as:

```
#!/bin/bash
$HOST="vm-qads-ilm"
$ILMD="$HOST.doc.ic.ac.uk"
$LOGIN="saspacey@shell1.doc.ic.ac.uk"

export ILOG_LICENSE_FILE=./access.ilm
hostname $HOST
ssh -f -L 3000:$ILMD:3000 $LOGIN sleep 10
./bin/x86_debian4.0_4.1/cplex
```

Both of the code snippets above start CPLEX running interactively, if you close your shell, your CPLEX process will be killed and any solution currently being calculated will not complete. You can avoid CPLEX terminating when you close your shell by wrapping your CPLEX process with:

```
screen ./bin/x86_debian4.0_4.1/cplex
```

then press `CTRL+A CTRL+D` to leave the process running in the background when you want to log-out and when you log-in again type `screen -r` to return to CPLEX. Another way to run CPLEX when you are not logged-in is as a perpetual background process which is discussed further at the end of the next section.

2. SOLVING PROBLEMS WITH CPLEX

You can create problems for CPLEX to solve using simple text files in Linear Programming (LP) format. Here is a simple optimisation problem in LP format:

```
MINIMIZE
    obj: 5.8 x_1 + 3 x_2

SUBJECT TO
    r1: x_1 + 2.1 x_2 = 6
    r2:       3   x_2 < 4.2

BOUNDS
    x_1 >= 0
    x_2 >= 0

INTEGER
    x_1

END
```

In the above program `obj`, `r1` and `r2` are optional labels used in detailed solutions; the constraints and `BOUNDS` can be of any relational form (i.e. <=, >=, >, < or =); the variable `x_1` must be a positive `INTEGER` and `x_2` can be any positive real. It should be noted that there is no multiplier symbol between numbers and variable names, there has to be an `END` statement and all relations must have only numbers on the right hand side.

Two methods can be used to include binary variables in CPLEX LPs:

1. declare the variables as `INTEGER` with `BOUNDS` between 0 and 1.
2. remove the variables from both the `INTEGER` and `BOUNDS` sections and add a separate `BINARY` section.

the choice of binary representation method does not effect performance.

Assuming the LP file above is saved as `problem.lp`, you can solve it by typing:

```
read problem.lp
opt
```

at the interactive CPLEX command prompt. You should add the line `set parallel 1` before the `opt` command if you require result path reproducibility on multi-threaded CPLEX installs [2]. The optimal solution returned by CPLEX for the problem above is 26.057.

If you wanted to automate CPLEX to read and solve LPs, you could start with a script like `cplex.sh` below:

```
#!/bin/bash
$CPLEX="/opt/cplex11/cplex"

rm -f cplex.log 2> /dev/null
rm -f results.s 2> /dev/null
$CPLEX 2> _cplex.err <<CPLEX_CMD

read problem.lp
opt
write results.s sol all
quit

CPLEX_CMD
```

and run the script as a perpetual background process with:

```
nohup ./cplex.sh > /dev/null &
```

Aside from LP forms, optimisation problems can also be represented in the OPL [1], AMPL [3] and GAMS [4] modelling languages or passed from C, C++ or Java to CPLEX using one of the ILOG Concert Technology APIs [2]. The modelling languages all allow relational constraints like LPs but can abstract the general problem logic away from the specific problem instance data. The AMPL and GAMS modelling languages have the added advantages of being portable across a range of different solvers and of being able to represent both concave and convex problems [5].

3. OBTAINING DETAILED SOLUTION RESULTS

To obtain the variable values that correspond to a solution you can save the detailed CPLEX results to a file with:

```
write results.s sol all
```

this saves solution variable values, slacks and objectives in an XML format which is easy to analyse.

In the XML file, your optimal result will be tagged with:

```
solutionName="incumbent"
```

Note that CPLEX 11 can report solutions in the `results.s` file with a lower objective than the final "incumbent". If you see this it is a bug, but you should extract the variable settings from the solution, set them as constraints in your model and check the objective value to confirm.

You should save the CPLEX log file `cplex.log` with your results, original problem formulation and CPLEX version and settings information for future reference when quoting results.

4. BIG-MS AND OBJECTIVE SENSITIVITY

When creating a Linear Program with logic relations like those of section 9, it is often necessary to use large constant multipliers generally called big-M's which when coupled with your variable and objective values contribute to the range of numbers in your problem. If you have too large a range of numbers in your problem not only can your solution times suffer, but your solution values can actually become invalid as demonstrated in table 1.

M_0 Multiple	Objective Error
1	0%
10	0%
100	1%
1000	60%
10000	100%

Table 1: Errors in the reported best solution objectives found by CPLEX 11 when compared to the objective of the true optimal solution for a GQMIP [6] problem using different multiples of a nominal big-M constant M_0.

Often you can reduce your number range by scaling the problem objective and variables and setting big-M's on a constraint by constraint basis rather than say using the standard C programmers default of UINT_MAX or the theoretical limit of [7]. Alternatively you could consider converting your big-M constraints to CPLEX indicators [2] or using the GNU GLPK [8] solver GLPSOL instead of CPLEX with the exact arithmetic option:

```
glpsol --exact --cpxlp ./problem.lp
```

however both of these alternatives are likely to have a detrimental impact on your overall solution times [9].

5. SOLUTION TIMES

All Linear Programs with only real variables can be solved in polynomial time [10]. However, the same is not true for Mixed Integer Linear Programs (MILPs) with integer or boolean variables which are often combinatorial in nature.

CPLEX uses Branch and Bound [11] to reduce the MILP search space and can solve MILPs quickly *provided the relaxed problem has sufficient sensitivity*. If you find your MILPs are taking too long to solve, you will need to reformulate your model or create your own bounding relaxations and, for example, integrate them into CPLEX's Branch and Bound algorithm using the ILOG Java Concert Technology API [2].

The effect of model formulation can be seen in table 2 where three different MILP forms are used to solve the same optimisation problem and produce markedly different Branch and Bound tree sizes as a result of their different lower bound relaxation granularities.

Method	B&B Nodes
$GQAP$	39
mms	64
mxs	1,635,667

Table 2: Branch and Bound search tree nodes for three equivalent versions of a GQAP problem solved with default options in CPLEX 11. $GQAP$ is standard GQAP [12] and mms and mxs are optimistic and robust GQMIP [6] forms respectively. All three forms produce the same objective.

6. QUICK SOLUTIONS

CPLEX includes a local neighbourhood heuristic search algorithm and other options that can quickly approach an optimal solution in cases where combinatorial complexity can not be avoided. The CPLEX heuristics slow down the overall solution process but can often produce better results than bespoke heuristic algorithms after only a few seconds of execution.

The CPLEX neighbourhood heuristic and solution focus can be set with the options:

```
set mip strategy rinsheur 100
set mip strategy probe 3
set mip cuts all 2
set emphasis mip 3
```

before the `opt` command in either the interactive solver or the `cplex.sh` script from section 2.

You can suspend the CPLEX solution process at any time by pressing `CTRL+C` in the interactive shell. This allows you to save an intermediary solution or change the CPLEX options and continue the optimisation process by typing `opt` again. For example, to save an intermediary solution after the CPLEX upper bound stabilises and then reset the RINS heuristic for faster completion press `CTRL+C` and type:

```
write results_quick.s sol all
set mip strategy rinsheur 100000
opt
```

In a script you can prematurely terminate CPLEX (without the option to continue) with the setting:

```
set timelimit <seconds>
```

In the interest of completeness, you should also be aware of the CPLEX MIP tolerance options `uppercutoff` and `lowercutoff` which, while apparently attractive for speeding up solutions in the presence of known bounds, do not actually speed-up the Branch and Bound process when relaxed bounds are loose.

7. START FILES

You can export the current CPLEX results for use as a starting point for future runs with:

```
write results.s sol all
```

To read in the file as a CPLEX solution starting point use:

```
read results.s mst
```

It should be noted that CPLEX strictly only needs a MST file to restart from a previous solution [2], however the full results file produced by the first statement above is more generally useful than the compact minimal MST start files which do not, for example, contain objective information.

8. MEMORY LIMITS

As CPLEX works through a combinatorial search tree it keeps track of branches taken and bounding result information at each tree node. If your problem can not be easily cut by Branch and Bound, the search tree can be combinatorial in size and can cause CPLEX to crash or at least stop with an error if your physical memory limit is exceeded.

To avoid physical memory limits, use the following options to allow CPLEX to uses the disk to store large trees:

```
set workmem 256
set mip strategy file 3
```

As a rule of thumb, the `workmem` value should not be more than half your physical memory (in megabytes) to ensure CPLEX has enough memory for code and intermediaries and that the OS does not need to resort to paging given other resident applications.

As CPLEX runs, you may see your free memory drop well below the 50% mark recommended above. This may be because of OS buffers on the CPLEX tree files which the OS will free automatically as memory is needed.

9. LINEAR PROGRAMMING LOGICAL EQUIVALENCIES

I conclude this paper with a set of lemmas I developed to demonstrate how logical functions that Computer Science students will already be aware of can be expressed as Integer Linear Programming (ILP) minimisation constraints. These lemmas and relations can be used as a starting point for mathematically modelling a logical problem.

Lemma 9.1. *Boolean logical NOT* ($\gamma = \neg\alpha \ : \ \alpha, \gamma \in \mathbb{B}$) *can be expressed as linear programming constraints through:*

$$(1-\alpha) \leq \gamma \leq (1-\alpha) \tag{1}$$

Proof. Trivial. □

Lemma 9.2. *Boolean logical AND* ($\gamma = \alpha \wedge \beta \ : \ \alpha, \beta, \gamma \in \mathbb{B}$) *can be expressed as linear programming constraints through:*

$$\alpha + \beta - 1 \leq 2\gamma \leq \alpha + \beta \tag{2}$$

Proof. Proof is through the logic table below.

α	β	$\alpha+\beta-1$	$\alpha+\beta$	γ
0	0	-1	0	0
0	1	0	1	0
1	0	0	1	0
1	1	1	2	1

□

Corollary 9.3. *Boolean logical AND is equivalent to quadratic multiplication in* $\mathbb{B}$ *and so lemma 9.2 represents a basis for the linearisation of quadratic constraints.*

Lemma 9.4. *Boolean logical OR* ($\gamma = \alpha \vee \beta \ : \ \alpha, \beta, \gamma \in \mathbb{B}$) *can be expressed as Linear Programming constraints through:*

$$\alpha + \beta \leq 2\gamma \leq 2(\alpha + \beta) \tag{3}$$

Proof. Proof is through the logic table below.

α	β	$\alpha+\beta$	$2(\alpha+\beta)$	γ
0	0	0	0	0
0	1	1	2	1
1	0	1	2	1
1	1	2	4	1

□

Lemma 9.5. *The 0-1 threshold of an integer parameter* ($\beta = \min(x, 1) \ : \ x \in \mathbb{N}^*, \beta, \gamma \in \mathbb{B}$) *can be expressed as linear programming constraints through:*

$$x \leq M\beta \leq Mx \tag{4}$$

Proof. Proof is through the logic table below.

x	Mx	β
0	0	0
1	M	1
2	$2M$	1
n	nM	1

□

ACKNOWLEDGEMENTS

Dr D. Kuhn and W. Wiesemann provided valuable contributions and advice for the construction of this paper.

REFERENCES

[1] ILOG Inc. `http://www.ilog.com`

[2] CPLEX 11.2 Manuals *ILOG Inc.*, (2008).

[3] AMPL Optimization LLC `http://www.ampl.com`

[4] GAMS Development Corp. `http://www.gams.com`

[5] ROCKAFELLAR, R.T. Convex Analysis. *Princeton University Press*, (1970).

[6] SPACEY, S.A. Computational Partitioning for Heterogeneous Architectures *Imperial Ph.D. Thesis*, (2009).

[7] PAPADIMITRIOU, C.H., STEIGLITZ, K. Combinatorial Optimization: Algorithms and Complexity. *Dover Publications Inc.*, New York, (1998).

[8] GLPK (GNU Linear Programming Kit) *Free Software Foundation*, `http://gnu.org/software/glpk`

[9] SCIP: Solving Constraint Integer Programs *Zuse Institute Berlin (ZIB)*, `http://scip.zib.de`

[10] KARMARKAR, N. A New Polynomial Time Algorithm for Linear Programming *Combinatorica*, Vol. 4, No. 4, pp 373-395, (1984).

[11] WOLSEY, L.A. Integer Programming. *John Wiley & Sons Inc.*, (1998).

[12] LEE, C., MA Z. The Generalized Quadratic Assignment Problem. *University of Toronto*, (2004).