Chapter 2: First steps with or-tools

Contents

1 Content

We will start by testing that your installation of the or-tools library is operational in section 2. After being relieved (phew, it does work, does it?), we will write our first *hello world* codes in section 3 and we'll learn the basic functionnalities of the CP solver. In section 4, we propose a better model to solve the cryptarithmetic puzzle we solved in the manual. This model will be used to create a cryptarithmetic puzzle generator in section 5.

2 Is my installed or-tools library operational?

In the section 2.1 of the manual, you can find how to download and install the *or-tools library*. If everything goes smoothly, you should be able to compile the next code:

```
#include "constraint_solver/constraint_solver.h"

int main(int argc, char **argv) {
  operations_research::Solver solver("First try!");
  return 0;
}
```

Mission 2.1

Grab your favorite text editor/IDE (Integrated Development Environment) and write the code above in a file named `first_try.cc`. Open the provided `Makefile` and edit the variable `OR_TOOLS_TOP` to the top main directory of your *or-tools library*. Invoke the makefile with the associated `TARGET` variable set to `first_try`:

```
make cpexe TARGET=first_try
```

End Mission 2.1

Were you able to compile and link the code? If not, retry installing the *or-tools library*. If you face too many difficulties, ask for help on the mailing list (http://groups.google.com/group/or-tools-discuss).

3 Hello world

For our first real code, we will ask the solver to decide between two variables each with a $0-1$ domain. We want the solver to assign them different values. Our first try will be to use a simple linear model:

$$a + b \leqslant 1$$
$$a + b \geqslant 1$$

with $a, b \in \{0, 1\}$. We will do this step by step.

Mission 2.2

Open a file named `hello_world1.cc`. As explained in subsection 2.3.2 of the manual, we need some headers:

```
#include <vector>

#include "base/logging.h"
#include "constraint_solver/constraint_solver.h"
```

Our preferred way to code is to nest our functions in the `namespace operations_research` (see the subsection 2.3.3 of the manual):

```
namespace operations_research {

void Hello_World() {
  // Constraint programming engine
  Solver solver("Hello World!");
  ...
}
}   // namespace operations_research

// ----- MAIN -----
int main(int argc, char **argv) {
  operations_research::Hello_World();
  return 0;
}
```

Now, you know how to compile and link this file:

```
make cpexe TARGET=hello_world1
```

Because the domain of our two variables is $\{0, 1\}$, we can use the nifty `MakeBoolVar()` factory method. Don't forget the remark about factory methods we made in the subsection 2.3.5 of the manual.

```
  // Decision variables
  IntVar* const a = solver.MakeBoolVar("A");
  IntVar* const b = solver.MakeBoolVar("B");

  // We need to group variables in a vector to be able to create
  // the DecisionBuilder
  std::vector<IntVar*> vars;
  vars.push_back(a);
  vars.push_back(b);
```

Our two constraints don't represent any challenge:

```
  // Constraints
  solver.AddConstraint(solver.MakeLessOrEqual(solver.MakeSum(a,b)->Var(),1));
  solver.AddConstraint(solver.MakeGreaterOrEqual(solver.MakeSum(a,b)->Var(),1));
```

Do you remember why we **must** call the `Var()` method? If not, see the subsection 2.3.7 of the manual about constraints.

To define the search, we need a `DecisionBuilder`:

```
  DecisionBuilder* const db = solver.MakePhase(vars,
                                               Solver::CHOOSE_FIRST_UNBOUND,
                                               Solver::ASSIGN_MIN_VALUE);
```

Next, we start the search and ask for the values of the solution:

```
solver.NewSearch(db);

if (solver.NextSolution()) {
  LOG(INFO) << "Solution found:";
  LOG(INFO) << "A=" << a->Value() << " " << "B=" << b->Value();
} else {
  LOG(INFO) << "Cannot solve problem.";
}  // if (solver.NextSolution())

solver.EndSearch();
```

If you are not sure what we are doing here, you can refresh your memory by reading subsection 2.3.9 of the manual.

End Mission 2.2

For our second try, let's use a non linear constraint:

$$ab + a = 1.$$

Mission 2.3

Implement the non linear constraint in a file named `hello_world2.cc`.

End Mission 2.3

⋆ Mission 2.4

What happens if you change the constraint to $ab + a = 0$ and why?

End Mission 2.4

4 A better model to solve cryptarithmetic puzzles

In subsubsection 2.2.2 of the manual, to solve

```
    C P
+   I S
+ F U N
-------
T R U E
```

we proposed a model with only two constraints: one linear constraint to represent the equality between the two sums and the `AllDifferent` constraint to ensure that all the letters represent different digits. A better model consists in several linear equalities, one for each column. Indeed, this equality can be distributed among the columns. If we use a general base b as in the manual, the equality becomes

$$\begin{array}{rrrrrrr} & & & + & \mathtt{C} \cdot b & + & \mathtt{P} \\ & & & + & \mathtt{I} \cdot b & + & \mathtt{S} \\ & & + & \mathtt{F} \cdot b^2 & + & \mathtt{U} \cdot b & + & \mathtt{N} \\ \hline = & \mathtt{T} \cdot b^3 & + & \mathtt{R} \cdot b^2 & + & \mathtt{U} \cdot b & + & \mathtt{E} \end{array}$$

To distribute the sum among the columns, we must be carreful as the sums in the columns can "overflow" in the next column, i.e. we must use carry variables. For instance, in the last column, $\mathtt{P} + \mathtt{S} + \mathtt{N}$ might be greater than $\mathtt{E}$ in base b. Therefore, to ensure the equality, we must add a carry variable $\mathtt{C}_1$:

$$\mathtt{P} + \mathtt{S} + \mathtt{N} = \mathtt{E} + \mathtt{C}_1.$$

⋆ Mission 2.5

Develop and implement a model that uses carry variables to solve the cryptarithmetic puzzle `CP + IS + FUN = TRUE` in any base b. The base is given as a command line parameters (use the Google gflags library as explained in subsection 2.5.1 of the manual). Try to use the smallest possible domains for the carry variables. Collect also all the solutions with a `AllSolutionCollector` (see section 2.4 of the manual) and print them.

End Mission 2.5

5 How to generate a cryptarithmetic puzzle?

Unlike in our example, a real cryptarithmetic puzzle should only have one solution.

⋆⋆ Mission 2.6

Write a program that generates real cryptarithmetic puzzles in any base b with the help of the CP solver. Use two text files containing each a list of word, one per line. The first file provides the words you'll use in your addition (you can choose another operation if you wish). A parameter m indicates the number of words to use. The second file contains the result of the addition (operation). Generate all existing cryptarithmetic puzzles (with a unique solution).

For instance, if the number of words is $m = 2$, the base $b = 10$, the first file contains the words `CRASH` and `HACKER` and the second file contains the word `REBOOT`, you program should generate the cryptarithmetic puzzle `CRASH + HACKER = REBOOT`.

Don't try to create a clever algorithm, choose an elegant and simple solution. Limit the search time using a command line parameter. If you don't remember how to limit the search time, read subsubsection 2.5.2 of the manual.

End Mission 2.6