An Introduction to SCIP

Cornelius Schwarz
University of Bayreuth
cornelius.schwarz@uni-bayreuth.de

September 28, 2010

1 Preface

In this tutorial we give a short introduction to the constraint integer programming framework SCIP (solving constraint integer programs) which was developed by Tobias Achterberg [1] as part of his PhD thesis. SCIP combines constraint programming and mixed integer programming (MIP) into one framework. We will focus on the mixed integer part here. We show how to use SCIP as a MIP solver backend using the n-queens example. This introduction is currently adapted to SCIP 2.0 and will probably be extended to the development of plugins in the future.

SCIP is free for academical use, but you will also need an LP solver. For this purpose you can use SoPlex, which was developed by Roland Wunderling [2] as part of his PhD thesis and is distributed under the same conditions as SCIP. The easiest way is to download the ZIB Optimization Suite, which contains SCIP/SoPlex and the mathematical modeling language Zimpl [3].

A good starting point for SCIP is to look at the examples bundled with the source code. As a reference you can use the doxygen documentation, which is available online at `http://scip.zib.de/doc/html/index.html`. You should start by looking at "File List → scip.h". Here you find the most important functions, sorted by categories like "Global Problem Methods", "Variable Methods", "Constraint Methods" and so on. Another place to look at is "File Members → All → s. Here you also find functions like `SCIPcreateConsLinear` which is not defined in "scip.h" – since it is a plugin – but in "cons_linear.h". The most useful SCIP functions start with the prefix "SCIP".

2 Using SCIP as a MIP solver backend

In this section we show how to use SCIP as a backend for solving mixed integer programs by developing a solver for the n-queens problem. We first give a brief introduction into the problem and than describe a C++ program for solving it. The model is based on the one described in the Zimpl documentation.

2.1 The n-queens problem

The n-queens problem asks how to place n queens on an $n \times n$ chess board in a way that no two queens interfere. In detail this means:

- In each vertical line of the board only one queen is allowed, we will refer to these lines as columns.
- In each horizontal line of the board only one queen is allowed, these lines will be called rows later on.
- In each diagonal line only one queen is allowed.

This can be modeled as a binary program in the following way: Let $x_{i,j} \in \{0,1\}$ denote whether a queen is placed on the ith row and the jth column of the chess board. Since the problem is to find a placement, the objective function is irrelevant. We add, however, the redundant objective to maximize the number of placed queens:

$$\max \sum_{i=1}^{n} \sum_{j=1}^{n} x_{i,j}$$

Now we force exactly one queen to be placed in every column and every row:

$$\begin{aligned} \sum_{i=1}^{n} x_{i,j} &= 1,\ j = 1, \ldots, n \\ \sum_{j=1}^{n} x_{i,j} &= 1,\ i = 1, \ldots, n \end{aligned}$$

The diagonal rows are a little bit more complicated to write up:

$$\begin{aligned} \sum_{i=1}^{n-j+1} x_{i,j+i-1} &\leq 1,\ j = 1, \ldots, n \\ \sum_{j=1}^{n-i+1} x_{i+j-1,j} &\leq 1,\ i = 1, \ldots, n \\ \sum_{i=1}^{n-j+1} x_{i,n-j-i+2} &\leq 1,\ j = 1, \ldots, n \\ \sum_{j=1}^{n-i+1} x_{i+j-1,n-j+1} &\leq 1,\ i = 1, \ldots, n \end{aligned}$$

2.2 Error handling in SCIP

Before we transform the n-queens IP program into a SCIP program, we first consider a general point when working with SCIP functions: Most SCIP functions return a value of the type `SCIP_RETCODE`. If this is equal to `SCIP_OKAY`, then everything went well, otherwise it indicates an error code. Therefore the normal call of a SCIP function returning a `SCIP_RETCODE` (we use `SCIPfunction` as a generic name – replace it with whatever function you are calling) is

```
SCIP_RETCODE retcode;
retcode = SCIPfunction();
if (retcode != SCIP_OKAY)
{
```

```
   // do your error handling here
}
```

Since this is a lot of code for every function call, SCIP provides two macros namely `SCIP_CALL` and `SCIP_CALL_ABORT`. The second one just aborts the execution by calling `abort()` if an error occured. The first one calls the SCIP function and, in the error case, returns the retcode. This results in the following code:

```
SCIP_RETCODE myfunction(void)
{
   SCIP_CALL(SCIPfunction());
   SCIP_CALL(SCIPotherfunction());
}
int main(int args, char * argv)
{
   SCIP_RETCODE retcode = myfunction();
   if (retcode != SCIP_OKAY)
   {
      // do your error handling here
   }
}
```

While this is nice for C programs, there is a problem when using `SCIP_CALL` from C++: A C++ constructor is not allowed to return a value. The same is true for destructors. Therefore we supply a third method, the `SCIP_CALL_EXC` macro. This behaves just like `SCIP_CALL`, but instead of returning the error code it throws an exception of a new type `SCIPException`. So the example above would now be written as:

```
int main(int args, char * argv)
{
   try
   {
      SCIP_CALL_EXC(SCIPfunction());
      SCIP_CALL_EXC(SCIPotherfunction());
   } catch(SCIPException & exec)
   {
      cerr<<exec.what()<<endl;
      exit(exec.getRetcode());
   }
}
```

2.3 Include files

For a SCIP based project there are three main header files to consider. The first and most important one is of course “scip/scip.h”. It declares the `SCIP` pointer and all public functions. You may have noticed that SCIP can be extended by plugins. In fact most parts of the MIP solver like heuristics, separators, etc. are implemented as plugins. To use them, include “scip/scipdefplugins.h”.

These two header files are C type. In early versions of SCIP it was necessary to wrap them in an `extern "C"` statement. As of version 1.1 SCIP now detects a C++ compiler and adds `extern "C"` own its own.

The last header file to consider is "objscip/objscip.h" if you want to use the C++ wrapper classes distributed with SCIP. For the queens example we do not develop own plugins, so we just use

```
#include <scip/scip.h>
#include <scip/scipdefplugins.h>
```

2.4 Developing a queens solver

When you use SCIP you have to do the following steps:

- initialize the SCIP environment
- load all desired plugins (including your own, if you like)
- create a problem
- add variables and constraints to the problem
- solve the problem
- access results
- free the SCIP environment

You can of course cycle through some of these steps like accessing the results, modifying the problem and solving again. We will now describe these steps in more detail for the queens solver.

2.4.1 Initializing the SCIP environment

In this section, we start developing our queens solver. Before you can do anything with SCIP, you have to create a valid `SCIP` pointer. For this purpose use the `SCIPcreate` function:

```
SCIP* scip;
SCIP_CALL_EXC(SCIPcreate(& scip));
```

2.4.2 Loading all desired plugins

After we created our `SCIP` pointer we load the plugins. In SCIP nearly everything is a plugin: heuristics, separators, constraint handlers, etc. Whenever you want to use one you first have to include it. This is done by various `SCIPinclude` functions like `SCIPincludeHeur` for heuristics or `SCIPincludeConshdlr` for constraint handlers. This also activates the default display plugins which writes various messages to standard output. (If you do not like this you can disable it by a call of `SCIPsetMessagehdlr(NULL)`.) All together we get:

```
SCIP_CALL_EXC(SCIPincludeDefaultPlugins(scip));
// SCIP_CALL_EXC(SCIPsetMessagehdlr(NULL);
// uncomment the above line to disable output
```

2.4.3 Creating a problem

Now we can create the IP model for the queens solver in SCIP. First we create an empty problem with `SCIPcreateProb`. The first argument is our `SCIP` pointer and the second is the name of the problem. You can also supply user specific problem data and call back functions to handle them, but normally you will not need them and can safely set them to `NULL`:

```
SCIP_CALL_EXC(SCIPcreateProb(scip, "queens", NULL, NULL,
                             NULL, NULL, NULL, NULL, NULL));
```

The default objective sense for SCIP problems is minimizing. Since we have a maximization problem we have to change this:

```
SCIP_CALL_EXC(SCIPsetObjsense(scip, SCIP_OBJSENSE_MAXIMIZE));
```

2.4.4 Creating variables

Now it is time to fill the empty problem with information. We start by creating variables, one binary variable for every field on the chess board. Variables are accessed through the type `SCIP_VAR*`. Associated with each variable is a type (continuous, integer, or binary), lower and upper bound and a objective. In our case, the type is binary for all variables, the lower bound is naturally 0, the upper bound 1, and the objective is 1 for all variables:

```
SCIP_VAR* var;
SCIP_CALL_EXC(SCIPcreateVar(scip, & var, "x#i#j", 0.0, 1.0, 1.0,
                            SCIP_VARTYPE_BINARY, TRUE, FALSE,
                            NULL, NULL, NULL, NULL, NULL));
```

Here, you should replace i and j by the actually row and column number of the variable. The fourth argument is the lower bound, the fifth the upper bound, the sixth the objective, and the seventh the type. After that you specify two boolean parameters indicating whether this variable is in the initial (root) LP and whether it is allowed to be removed during aging. Like in `SCIPcreateProb` you can use the last five parameters to specify user data. We set these parameters to `NULL`. After creating the `SCIP_VAR` pointer it is time to add it to the SCIP problem:

```
SCIP_CALL_EXC(SCIPaddVar(scip, var));
```

You should store the `SCIP_VAR` pointers somewhere, since you will need them to add the variables to constraints and to access their values in the final solution and so on. In our example, you can use a two dimensional STL vector for that purpose.

2.4.5 Creating constraints

Creating and adding variables is just half of the battle. To ensure feasibility, we have to add the constraints we described above. To create a constraint in SCIP you first need to specify a constraint handler. The constraint handler is responsible for checking feasibility, tighten variable bounds, adding new rows to the underlying LP problem and so on. The creating method depends on the

actual constraint you want to use and is usually called `SCIPcreateConsName` – for instance `SCIPcreateConsLinear`. Although there are many different default constraints like knapsack, logic-OR, etc., it is a safe way to create them as linear constraints. The presolver will automatically transform them to the right constraint type. We will therefore add all our constraints as type linear and describe the handler here.

The linear constraint handler handles constraint of the following type:

$$lhs \leq a^T x \leq rhs$$

There are three special cases of these: For equality constraints set $lhs = rhs$, for lesser equal constraints, set $lhs = -$`SCIPinfinity(scip)` and for greater equal constraints $rhs =$ `SCIPinfinity(scip)`. So the creating of the diagonal constraints looks as follows:

```
SCIP_CONS* cons;
SCIP_CALL_EXC(SCIPcreateConsLinear(scip, & cons, "diag",
                                   0, NULL, NULL, 0, 1.0, TRUE,
                                   TRUE, TRUE, TRUE, TRUE, FALSE,
                                   FALSE, FALSE, FALSE, FALSE);
```

The first is, as usual, the `SCIP` pointer and the second the `SCIP_CONS` pointer, which allows to access the constraint later. After that you can specify variables to be added to the constraint. This could be done by specifying the number, an array of `SCIP_VAR` pointers to variables, and an array of values of the coefficients, stored as doubles. We skip the adding at this point and use the function `SCIPaddCoefLinear` described later on. The next two entries are lhs and rhs. In our cases 0 and 1. Then you specify the following parameters:

initial set this to `TRUE` if you want the constraint to occur in the root problem

separate set this to `TRUE` if you would like the handler to separate, e. g. generate cuts

enforce set this to `TRUE` if you would like the handler to enforce solutions. This means that when the handler declares an LP or pseudo solution as infeasible, it can resolve infeasibility by adding cuts, reducing the domain of a variable, performing a branching, etc.

check set this to `TRUE` if the constraint handler should check solutions

propagate set this to `TRUE` if you want to propagate solutions, this means tighten variables domains based on constraint information

local set this to `TRUE` if the constraint is only locally valid, e. g., generated in a branch and bound node

modifiable set this to `TRUE` if the constraint may be modified during solution process, e. g. new variables may be added (colum generation)

dynamic set this to `TRUE` if this constraint is subject to aging, this means it will be removed after being inactive for a while (you should also say `TRUE` to removable in that case)

removable set this to `TRUE` to allow the deletion of the relaxation of the constraint from the LP

stickingatnode set this to `TRUE` if you want the constraint to be kept at the node it was added

Variables which are not added at the creation time of the constraint can be added by calling:

```
SCIP_CALL_EXC(SCIPaddCoefLinear(scip, cons, var, 1.0));
```

Here "1.0" is the matrix coefficient.

2.4.6 Solving the problem

When the problem is setup completely we can solve it. This is done by

```
SCIP_CALL_EXC(SCIPsolve(scip));
```

SCIP then starts transforming and preprocessing the problem. After that it enters the solving stage where the root LP is solved, heuristics are run, cuts are generated, and the branching process starts. All plugins you wrote (heuristics, separators, etc.) will be called by SCIP through call back functions in this stage.

2.4.7 Accessing results

Now that the problem is solved, we want to know the solution data. Whether the problem has been solved to optimality, only feasible solutions were found, and so on, can be queried by `SCIPgetStatus`. We ignore this in our queens solver and start with the best solution found so far. This can be accessed by

```
SCIP_SOL* sol = SCIPgetBestSol(scip);
```

If SCIP did not find a solution `sol` is equal to `0`. Otherwise, you can get the objective value by `SCIPgetSolOrigObj`. In the queens example we want to know whether a queen is placed on a field or not. Therefore we need the value of the variable $x_{i,j}$ which can be accessed by `SCIPgetSolVal`. In the case of an integer or binary variable, care must be taken, because this functions returns double values. So if we want to query a binary variable we use the following:

```
if (sol == NULL)
{
   // output error message here and abort
}
if ( SCIPgetSolVal(scip, sol, var) > 0.5 )
{
   // value is one
}
else
{
   // value is zero
}
```

In this example, we of course use the knowledge that variables have 0/1 values only. There are special SCIP functions for performing numerical comparisons between values that are not known to be integer. For instance, you can use `SCIPisFeasEQ(scip, x, y)` for comparing whether x is equal to y within the feasibility tolerance of SCIP. This macro return true if $|x - y| < \epsilon$, where ϵ is the feasibility tolerance of SCIP (by default $\epsilon = 10^{-6}$).

2.4.8 Freeing the SCIP environment

Finally, we must free all the memory SCIP used. When we created the variables and constraints, the `SCIPcreateVar` and `SCIPcreateCons` captured the corresponding variables and constraints. This means that SCIP knows that we have a pointer to these and will only free the memory if we tell it that we do not need these pointers anymore. This is done by the `SCIPrelease` functions. So before we can free the `SCIP` pointer, we have to call:

```
SCIP_CALL_EXC(SCIPreleaseVar(scip, & var);
SCIP_CALL_EXC(SCIPreleaseCons(scip, & cons);
```

Then we close the SCIP environment:

```
SCIP_CALL_EXC(SCIPfree(& scip));
```

References

[1] Achterberg, Tobias: *Constraint Integer Programming*, PhD thesis, Technische Universität Berlin, 2007

[2] Wunderling, Roland: *Paralleler und Objektorientierter Simplex-Algorithmus*, PhD thesis, Technische Universität Berlin, 1996

[3] Koch, Thorsten: *Rapid Mathematical Programming*, PhD thesis, Technische Universität Berlin, 2004