Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

STEPHEN MAHER*, MATTHIAS MILTENBERGER*,
JOÃO PEDRO PEDROSO†, DANIEL REHFELDT*,
ROBERT SCHWARZ*, FELIPE SERRANO*

PYSCIPOPT: Mathematical Programming in Python with the SCIP Optimization Suite

* Zuse Institute Berlin, Germany, {maher, miltenberger, rehfeldt, schwarz, serrano}@zib.de
† Faculdade de Ciências da Universidade do Porto, Portugal, jpp@fc.up.pt

ZIB Report 16-64 (December 2016)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite

Stephen Maher Matthias Miltenberger João Pedro Pedroso
Daniel Rehfeldt Robert Schwarz Felipe Serrano

Abstract

SCIP is a solver for a wide variety of mathematical optimization problems. It is written in C and extendable due to its plug-in based design. However, dealing with all C specifics when extending SCIP can be detrimental to development and testing of new ideas. This paper attempts to provide a remedy by introducing PySCIPOpt, a Python interface to SCIP that enables users to write new SCIP code entirely in Python. We demonstrate how to intuitively model mixed-integer linear and quadratic optimization problems and moreover provide examples on how new Python plug-ins can be added to SCIP.

1 Introduction

Since its initial release in 2005, SCIP has matured into a powerful solver for various classes of optimization problems and has achieved considerable acclaim in academia and industry. It is distributed as part of the SCIP Optimization Suite [14], along with the LP solver SoPlex [6, 16], the modeling language ZIMPL [18], the generic column generation solver GCG [11], and the parallelization framework UG [17]. For an in-depth description of SCIP and the Optimization Suite in general we refer to the original publication [1] and the report about the latest release 3.2 [3]. SCIP is available in source code and provides tutorials and comprehensive documentation for researchers and practitioners on its web page [14], thereby allowing users to extend its functionality and write custom plug-ins. As yet, however, such extensions required not only knowledge of the C programming language, but furthermore impeded fast prototyping, an obstacle keeping some (potential) users from implementing their ideas within a reasonable amount of time. To overcome these impediments, we have developed PySCIPOpt [13], a Python interface to SCIP that allows for fast prototyping of new algorithmic concepts and concurrently benefits from the underlying high performance C code. The interface is implemented by means of the programming language Cython [8] and documented with the Python standard Docstring [10].

Finally, it is certainly worth mentioning that with JuMP [4] there is already an optimization software package available for the Julia language that provides both fast implementation possibilities and high performance. However, this software does not yet fully support SCIP.[1]

2 Modeling

The Python interface supports an intuitive modeling syntax using linear and quadratic expressions. The following example shows how to build up and subsequently solve a small mixed-integer

[1] SCIP can already be used to solve models formulated in JuMP via AMPL's `nl` format [7]. Furthermore, there is an ongoing development effort to develop an interface that supports callbacks [15].

quadratic programming problem:

Mathematical formulation:

$$\begin{aligned} \text{minimize } & x + y \\ \text{subject to } & 2x + y^2 \geq 10 \\ & x, y \geq 0 \\ & x \in \mathbb{R} \\ & y \in \mathbb{Z} \end{aligned}$$

Python code:

```
from pyscipopt import Model
scip = Model()
x = scip.addVar('x', vtype='C')
y = scip.addVar('y', vtype='I')
scip.setObjective(x + y)
scip.addCons(2*x + y*y >= 10)
scip.optimize()
```

Most methods can be called with a small number of parameters, leaving it to the interface to fill in the remaining parameters with default values. If necessary, these parameters can be set in any order to provide a more flexible interface. This feature is exemplified by the `addVar()` method, which is used in the example above with only two specified parameters, while in fact six may be provided:

```
addVar(self,                # the SCIP model
       name = '',           # name of the variable
       vtype = 'C',         # variable type ('C', 'I', or 'B')
       lb = 0.0,            # lower bound
       ub = None,           # upper bound
       obj = 0.0,           # objective coefficient
       pricedVar = False    # is it a pricing candidate?
       )
```

In this way, a minimalistic and intuitive code can be designed without surrendering any functionality of the wrapped SCIP method.

3 Extending SCIP: Writing Plug-Ins in Python

Every plug-in supported by PySCIPOpt is encapsulated in a separate file that declares its interface methods to SCIP. The file defines the Python base class for the respective plug-in, including its callbacks. These callback definitions can be left empty for optional callbacks, but need to be implemented by the user in case of fundamental ones. In line with the overall approach of PySCIPOpt, the user can thereby implement customized plug-ins with minimal effort, but access to more intricate functionalities of SCIP is still provided.

3.1 Constraint Handler Example: TSP

In this section we show how to design a simple constraint handler that can solve the traveling salesman problem (TSP).

The following model for solving the TSP has been proposed by Dantzig–Fulkerson–Johnson (DFJ) [2]. Let $G = (V, E)$ be a complete, undirected graph with $V = \{1, \cdots, n\}$ being the vertex set and E the edge set. Furthermore, let c_{ij} be the weight of the edge (i, j). We associate with each edge (i, j) a variable x_{ij}, with $x_{ij} = 1$ if edge (i, j) is used in the solution and $x_{ij} = 0$

otherwise. Thereupon, the DFJ integer programming formulation can be stated as follows:

$$\text{minimize} \quad \sum_{i,j} c_{ij} x_{ij} \tag{1}$$

$$\text{subject to} \quad \sum_{j} x_{ij} = 2 \qquad \forall\, i \in \{1, \cdots, n\} \tag{2}$$

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1 \quad \forall\, S \subsetneq \{1, \cdots, n\},\ |S| \geq 2 \tag{3}$$

$$x_{ij} \in \{0, 1\} \qquad \forall\, i < j \tag{4}$$

The constraints (3) exclude subtours by imposing that for any proper subset S of the vertex set V such that $|S| \geq 2$ a solution cannot encompass a cycle within S. However, as there is an exponential number of subsets of V, it is impractical to specify all of these constraints. A possible approach is to iteratively solve the problem, starting without these constraints and after each solving round add constraints (3) violated by the current solution.

The constraint handler in our PySCIPOpt TSP implementation does not generate its own constraints, but instead SCIP is querying whether the current solution is feasible and in case it is not, how feasibility can be achieved. This is accomplished by setting the `needscons` flag to `False` when including the constraint handler into SCIP.

Our example[2] uses the external Python library `networkx` to compute the connected components of a graph. The constraint handler then adds the corresponding subtour elimination constraints as described above when called in the `consenfolp()` callback. An integer feasible solution that satisfies the first set of constraints will be checked for subtours in the `conscheck()` callback of the `TSPconshdlr` class. The `conslock()` callback sets up locks on the variables of the constraint and is required for a correct implementation. In a nutshell, it tells SCIP how the variables can be rounded without violating the constraint.

For more information about the different callbacks we refer to the SCIP documentation [14].

When adding constraints via linear or quadratic expressions we recommend to use the `quicksum()` function. This eliminates the overhead of intermediate creation/destruction of multiple expressions and instead sets up one instance that will be iteratively extended. A similar implementation is also provided in the Python interface of the commercial optimization software Gurobi [12].

4 Conclusion and Outlook

With PySCIPOpt we provide a SCIP based optimization tool that allows for fast, minimalistic and intuitive programing, while still having the more intricate functionalities of SCIP up its sleeve. We hope that the availability of such a device will help mathematical programming experts set up prototypes more efficiently and moreover allow less experienced users to more easily make use of the wide range of capabilities provided by SCIP. In this way, PySCIPOpt may also serve as coherent tool to make (undergraduate) mathematical optimization students familiar with the subject, as has already successfully been the case for instance at the University of Porto [9].

Future developments naturally encompass the extension of PySCIPOpt to cover even more functionalities provided by SCIP. Missing functions can be easily added by specifying their header

[2] A proper implementation of this constraint handler would implement the callback `conssepalp` to separate the LP solution. The callback `consenfolp` is called at the end of the node processing loop, where possibly several calls to the `conssepalp` callback of the different constraint handlers have already been made.

For reasons of space we provide a concise, although rather inefficient, implementation.

declaration in the `scip.pxd` file and defining a corresponding wrapper function in the class `Model` in the `scip.pyx` file. Of special interest in this context are additional general nonlinear programming methods [5]: Our objective is to make them blend in with the already existing features, while maintaining the underlying minimalistic design of PySCIPOpt.

5 Acknowledgements

The authors would like to thank the anonymous reviewers for helpful comments on the paper. The work for this article has been partly conducted within the *Research Campus Modal* funded by the German Federal Ministry of Education and Research (fund number 05M14ZAM).

tsp_example.py

```
import networkx
from pyscipopt import Model, Conshdlr, quicksum, SCIP_RESULT

EPS = 1.e-6

class TSPconshdlr(Conshdlr):

  def __init__(self, variables):
    self.variables = variables

  def find_subtours(self, solution=None):
    edges = []
    x = self.variables
    for (i,j) in x:
      if self.model.getSolVal(solution, x[i,j]) > EPS:
        edges.append((i,j))
    G = networkx.Graph()
    G.add_edges_from(edges)
    components = list(networkx.connected_components(G))
    return [] if len(components) == 1 else components

  def conscheck(self, constraints, solution, check_integrality,
                check_lp_rows, print_reason):
    if self.find_subtours(solution):
      return {"result": SCIP_RESULT.INFEASIBLE}
    else:
      return {"result": SCIP_RESULT.FEASIBLE}

  def consenfolp(self, constraints, n_useful_conss, sol_infeasible):
    subtours = self.find_subtours()
    if subtours:
      x = self.variables
      for subset in subtours:
        self.model.addCons(quicksum(x[i,j] for(i,j) in pairs(subset))
                           <= len(subset) - 1)
        print("cut: len(%s) <= %s" % (subset, len(subset) - 1))
      return {"result": SCIP_RESULT.CONSADDED}
    else:
      return {"result": SCIP_RESULT.FEASIBLE}

  def conslock(self, constraint, nlockspos, nlocksneg):
    x = self.variables
    for (i,j) in x:
      self.model.addVarLocks(x[i,j], nlocksneg, nlockspos)

def create_tsp(vertices, distance):
  model = Model("TSP")
  x = {}
```

```
  for (i,j) in pairs(vertices):
    x[i,j] = model.addVar(vtype = "B",name = "x(%s,%s)" % (i,j))
  for i in vertices:
    model.addCons(
      quicksum(x[j,i] for j in vertices if j < i) +
      quicksum(x[i,j] for j in vertices if j > i) == 2)
  conshdlr = TSPconshdlr(x) # set up conshdlr with all variables
  model.includeConshdlr(conshdlr, "TSP", "TSP subtour eliminator",
                        needscons=False)
  model.setObjective(quicksum(distance[i,j] * x[i,j]
                      for (i,j) in pairs(vertices)), "minimize")
  return model, x

def solve_tsp(vertices, distance):
  model, x = create_tsp(vertices, distance)
  model.optimize()
  edges = []
  for (i,j) in x:
    if model.getVal(x[i,j]) > EPS:
      edges.append((i,j))
  return model.getObjVal(), edges

def pairs(vertices):
  for i in vertices:
    for j in vertices:
      if i < j:
        yield (i,j)

def test_main():
  vertices = [1, 2, 3, 4, 5, 6]
  distance = {(u,v):1 for (u,v) in pairs(vertices)}
  for u in vertices[:3]:
    for v in vertices[3:]:
      distance[u,v] = 10
  objective_value, edges = solve_tsp(vertices, distance)
  print("Optimal tour:", edges)
  print("Optimal cost:", objective_value)

if __name__ == "__main__":
  test_main()
```

References

[1] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.

[2] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 3:393–410, 1954.

[3] G. Gamrath, T. Fischer, T. Gally, A. M. Gleixner, G. Hendel, T. Koch, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, S. Vigerske, D. Weninger, M. Winkler, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 3.2. Technical Report 15-60, ZIB, Takustr.7, 14195 Berlin, 2016.

[4] M. Lubin and I. Dunning. Computing in Operations Research using Julia. *INFORMS Journal on Computing*, 27(2):238–248, 2015.

[5] S. Vigerske and A. Gleixner. SCIP: Global Optimization of Mixed-Integer Nonlinear Programs in a Branch-and-Cut Framework. Technical Report 16-24, ZIB, Takustr.7, 14195 Berlin, 2016.

[6] R. Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996.

[7] AmplNLWriter.jl. `https://github.com/JuliaOpt/AmplNLWriter.jl`.

[8] Cython. `http://www.cython.org/`.

[9] Decision Support Methods, Universidade do Porto. `http://www.dcc.fc.up.pt/~jpp/mad/`.

[10] PEP 0257 – Docstring Conventions. `https://www.python.org/dev/peps/pep-0257/`.

[11] GCG: Generic Column Generation. `http://www.or.rwth-aachen.de/gcg/`.

[12] Gurobi Optimizer Reference Manual. `http://www.gurobi.com/`.

[13] PySCIPOpt. `https://github.com/SCIP-Interfaces/PySCIPOpt`.

[14] SCIP: Solving Constraint Integer Programs. `http://scip.zib.de/`.

[15] SCIP.jl. `https://github.com/ryanjoneil/SCIP.jl`.

[16] SoPlex: Sequential object-oriented simPlex. `http://soplex.zib.de/`.

[17] UG: Ubiquity Generator framework. `http://ug.zib.de/`.

[18] ZIMPL: Zuse Institute Mathematical Programming Language. `http://zimpl.zib.de/`.