Dippy – a simplified interface for advanced mixed-integer programming

By Dr Michael O'Sullivan, Dr Cameron Walker, Qi-Shan Lim, Dr Stuart Mitchell and Assoc Prof Ted Ralphs

April 2014
Report, University of Auckland Faculty of Engineering, no. ???
ISSN 1178-3680

Dippy – a simplified interface for advanced mixed-integer programming

Michael O'Sullivan, Qi-Shan Lim and Cameron Walker
Department of Engineering Science, University of Auckland

Stu Mitchell
Stuart Mitchell Consultants Ted Ralphs
Department of Industrial Engineering, Lehigh University

April 27, 2014

Abstract

The development of mathematical modelling languages such as AMPL, GAMS, Xpress-MP and OPL Studio have made it easier to formulate mathematical models such as linear programmes, mixed-integer linear programmes and non-linear programmes for solution in solvers such as CPLEX, MINOS and Gurobi. However, some models cannot be solved using "out-of-the-box" solvers, so advanced techniques need to be added to the solver framework.

Many solvers, including CPLEX, Symphony and DIP, provide callback functions to enable users to customise the overall solution framework. However, this approach involves either expressing the mathematical formulation in a low level language such as C++ or Java, or implementing a complicated indexing scheme to be able to track formulation components such as variables and constraints between the mathematical modelling language and the solver's callback framework.

In this paper we present Dippy, a combination of the Python mathematical modelling language PuLP and the open source solver DIP. Using Dippy, users can express their model in a straightforward modelling language and use callback functions to access the PuLP model directly. We discuss the link between PuLP and DIP and give examples of advanced solving techniques expressed simply in Dippy.

1 Introduction

Using a high-level modelling language such as AMPL, GAMS, Xpress-Mp or OPL Studio enables Operations Research practitioners to express complicated mixed-

integer linear programming (MILP) problems quickly and (reasonably) easily. Once defined in one of these high-level languages, the MILP problem can be solved using one of a number of solvers. However, for many MILP problems using solvers "out of the box" is only effective for small problem instances (due to the fact that MILP problems are NP-hard). Advanced MILP techniques are needed for large problem instances and, in many cases, problem-specific techniques need to be included in the solution process.

Both commercial solvers such as CPLEX and open source solvers such as Cbc, Symphony and DIP (all from the COIN-OR repository [LH03]) provide callback functions that allow user-defined routines to be included in the solution framework. However, to make use of the callback functions and develop the user-defined routines, the user must first create their MILP problem in a low-level language (C, C++ or Java for Cplex, C or C++ for Cbc, Symphony or DIP). While defining their problem, they need to create structures to keep track of appropriate constraints and/or variables for later use in their user-defined routines. For a MILP problem of any reasonable size and/or complexity, the problem definition in C/C++/Java is a major undertaking and, thus, a major barrier to the development of customised MILP frameworks by both practitioners and researchers.

Given the difficulty of defining a MILP problem in a low-level language, another alternative is to return to the high-level mathematical modelling languages (AMPL, GAMS, Xpress-MP, OPL Studio) or their open source equivalents (such as FLOPC++, GAMSlinks and PuLP) to define the MILP problem. Then, by carefully constructing an indexing scheme, constraints and/or variables in the high-level language can be identified in the low-level callback functions and advanced MILP techniques used. However, implementing the indexing scheme is possibly as difficult as simply using the low-level language to define the MILP problem in the first place and so combining high-level and low-level languages does not remove the barrier to solution development.

The purpose of the research presented here is the removal of the barrier that prevents easy experimentation with/customisation of advanced MILP solution frameworks. To achieve this aim we need to:

1. provide an straightforward modelling system so that users can quickly and easily describe their MILP problems;

2. enable the callback functions to easily identify constraints and variables in the solution framework.

PuLP already provides a straightforward modelling system for describing MILP problems. In our research we extended PuLP so that a MILP could be defined and solved using Decomposition for Integer Programming (DIP) [RG05] within a Python file. We also extended DIP so that routines defined within that Python file will be called by the DIP callback functions. Variable scope within Python means that these user-defined (Python) routines have complete knowledge of the original problem defined in PuLP. Any extra information required to generate cuts, determine branches or generate columns is provided by the DIP callback functions. This "glue" between PuLP and DIP overcomes the barrier to easy cus-

tomisation of DIP and provides both practitioners and students a straightforward interface for describing problems and customising their solution framework.

The rest of this article is structured as follows. In section 2 we provide an overview of the interface between PuLP and DIP, including a description of the callback functions available in Python from DIP. Then, section 3 contains the description and model definition for case studies we will use to demonstrate the effectiveness of Dippy for experimenting with advanced MILP techniques. Next, in sections 4-7 we describe how to customise the DIP framework to experiment with improvements to the MILP solution frameworks available in DIP.We conclude in section 8 where we discuss how this project enhances the ability of researchers to experiment with approaches for solving difficult MILP problems.

2 Combining DIP and PuLP

PuLP is a mathematical modelling language in Python. Using PuLP, a user can quickly and (reasonably) easily define a MILP problem and solve it using a variety of solvers including CPLEX, Gurobi and Cbc. As PuLP is built from Python objects it is relatively easy to extend PuLP to use other solvers such as DIP. For more details on PuLP see the PuLP project in the COIN-OR repository [LH03].

Decomposition for Integer Programming (DIP) [RG05] provides a framework for solving MILP problems using 3 different methods[1]: 1) branch and cut; 2) branch, price and cut; and 3) branch, decompose and cut. In this paper we will restrict our attention to 1) and 2).

Branch and cut uses the classic branch-and-bound approach for MILP combined with the cutting plane method for removing fractionality in the branch-and-bound nodes' linear programmes. This framework is the basis of many state-of-the-art MILP solvers including Gurobi and Cbc. DIP provides callback functions for generating user-defined cuts and checking the feasibility of a solution (with respect to the user-defined cuts) within branch and cut. DIP also provides a callback function for running a heuristic at each node of the branch-and-bound tree.

Branch, price and cut uses Dantzig-Wolfe decomposition to split a large MILP problem into a master problem and one or more subproblems. Branch and cut is used on the master problem. The subproblems solve a pricing problem, defined using the master problem dual values, to add new variables to the master problem.

For branch, price and cut, the cut generation and heuristic callback functions can be used, but extra callback functions enable the user to define their own routines for finding initial variables to include in the master problem and for solving the subproblems to generate new master problem variables. For details on the methods and callback functions provided by DIP see [RG05].

Dippy extends PuLP to use the DIP solver, but it also enables the user-defined routines accessed by the DIP callback functions to be implemented in the same

[1]The skeleton for a fourth method (branch, relax and cut) exists in DIP, but this method is not yet implemented.

Python file as the MILP model. Variable scope in Python means that any constraints or variables defined in the MILP model are easily accessible in the user-defined routines. Also, DIP is implemented so that the MILP problem is defined the same way whether branch and cut or branch, price and cut is being used, i.e., it hides the implementation of the master problem and subproblems. This makes it very easy to switch between methods when experimenting with solution methods.

In addition to the DIP callback functions (listed in §2.1), we have added another callback function that enables user-defined branching in DIP and so can be used in any of the solution methods within DIP.

2.1 Callback Functions

Advanced Branching We modified `chooseBranchVar` in the DIP source to become
`chooseBranchSet`. This makes it possible for the user to define:

- a *down* set of variables with (lower and upper) bounds that will be enforced in the down node of the branch; and,
- an *up* set of variables with bounds that will be enforced in the up node of the branch.

A typical variable branch on an integer variable x with integer bounds l and u and fractional value α can be implemented by:

1. choosing the down set to be $\{x\}$ with bounds l and $\lfloor \alpha \rfloor$;
2. choosing the up set to be $\{x\}$ with bounds of $\lceil \alpha \rceil$ and u.

However, other branching methods may use advanced branching techniques such as the one demonstrated in section 4. From DIP, `chooseBranchSet` calls `branch_method` in Dippy.

Customised Cuts We extended `generateCuts` (in the DIP source) to call `generate_cuts` in Dippy. This enables the user to examine a solution and generate any customised cuts as necessary. We also modified `APPisUserFeasible` to call `is_solution_feasible` in Dippy, enabling users to check solutions for feasibility with respect to customised cuts.

Customised Columns (Solutions to Subproblems) We extended `solveRelaxed` (in DIP) to call `relaxed_solver` in Dippy. This enables the user to utilise the master problem dual variables to produce solutions to subproblems (i.e., columns in the master problem) using customised methods. We also modified `generateInitVars` to call `init_vars` in Dippy, enabling users to customise the generation of initial columns for the master problem.

Heuristics We extended `APPheuristics` (DIP) to call `heuristics` (Dippy). This enables the user to define customised heuristics at each node in the branch-and-bound tree (including the root node).

2.2 Interface

The interface between Dippy (in Python) and DIP (in C++) is summarised in figure 1.

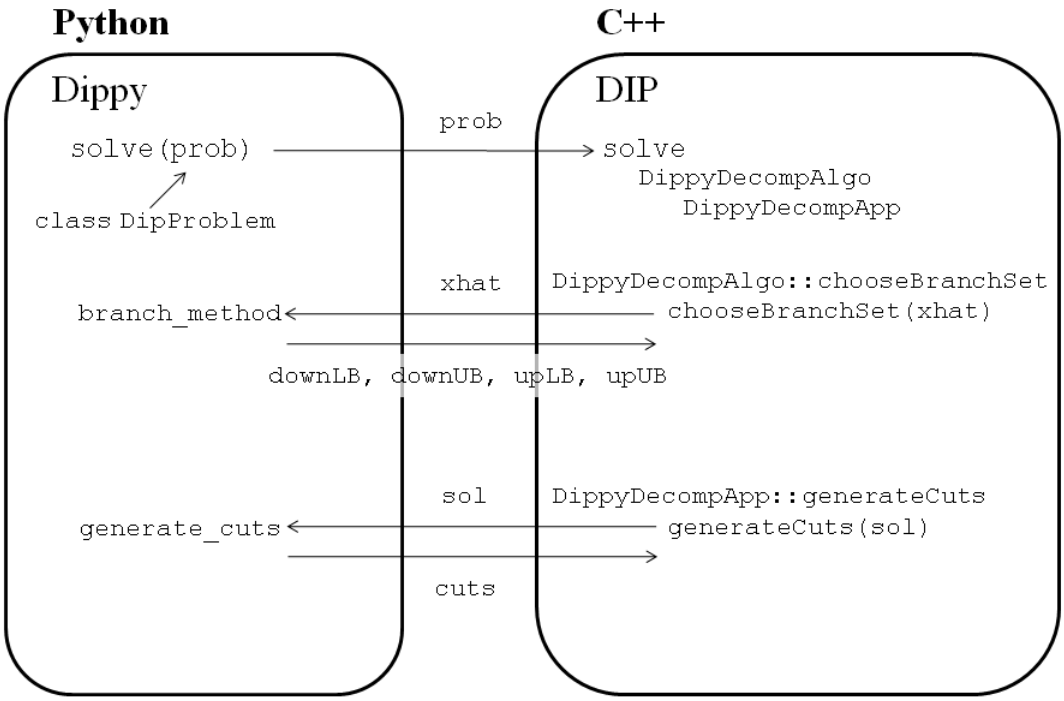

Figure 1: Interface between Dippy and DIP

The MILP is defined as a `DipProblem` and then solved using the `solve` command in Dippy, that passes the Python `DipProblem` object, `prob`, to DIP in C++. The DIP `solve` creates a `DippyDecompAlgo` object that contains a `DippyDecompApp` object, both of which are populated by data from `prob`. As the DIP `solve` proceeds branches are created by the `DippyDecompAlgo` object using `chooseBranchSet` which passes the current node's fractional solution `xhat` back to the `DipProblem` object `prob`'s `branch_method` function. This function generates lower and upper bounds for the "down" and "up" branches and returns to `DippyDecompAlgo::chooseBranchSet`. When DIP generates cuts, it uses the `DippyDecompApp` object's `generateCuts` function which passes the current node's solution `sol` to `prob`'s `generate_cuts` function. This function generates any customised cuts and returns a list of `cuts` back to

`DippyDecompApp::generateCuts`. These interfaces are replicated for the other callback functions provided by Dippy.

3 Case Studies

In this section we consider the case studies used to demonstrate the use of Dippy. The case studies are:

1. the capacitated facility location problem;
2. the coke supply chain problem (a capacitated facility location problem within a transshipment problem);
3. the travelling salesperson problem;
4. the cutting stock problem;
5. the wedding planner problem (a set partitioning problem)

In this section we will define the case studies in PuLP and demonstrate their solution in DIP without any customisation. Note that performance metrics throughout this paper were generated using Python 2.6.5 on a Dell Precision M4300 laptop with Intel Core 2 Duo CPU T9500@2.60GHz chipset and 777MHz, 3.50GB of RAM unless otherwise indicated. We used DIP version 0.8.7 and Dippy version 1.0.8.

3.1 The Capacitated Facility Location Problem (facility.py)

The solution of the capacitated facility problem determines where, amongst m locations, to place facilities and also assigns production of n products to these facilities in a way that (in this case study) minimises the wasted capacity of facilities. Each product $j = 1, \ldots, n$ has a production requirement r_j and each facility has capacity C. Extensions of this problem arise often in MILP in problems including network design and rostering.

The MILP formulation of the capacitated facility location problem is straightforward. The decision variables are

$$x_{ij} = \begin{cases} 1 & \text{if product } j \text{ is produced at location } i \\ 0 & \text{otherwise} \end{cases}$$

$$y_i = \begin{cases} 1 & \text{if a facility is located at location } i \\ 0 & \text{otherwise} \end{cases}$$

$$w_i = \text{“wasted” capacity at location } i$$