Dippy – A Simplified Interface to Advanced Integer Programming Techniques

Michael O'Sullivan, Qi-Shan Lim and Cameron Walker
Department of Engineering Science, University of Auckland
michael.osullivan@auckland.ac.nz

Stu Mitchell
Light Metals Research Centre, University of Auckland

Abstract

The development of mathematical modelling languages such as AMPL, GAMS, Xpress-MP and OPL Studio have made it possible to formulate mathethematical models such as linear programmes, mixed integer linear programmes and non-linear programmes for solution in solvers such as CPLEX, MINOS and Gurobi in a reasonable time frame. 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 and open source solvers such as 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 callbacks that access the PuLP model directly. We discuss the link between PuLP and DIP and give examples of advanced solving techniques expressed simply in Dippy.

Key words: Integer programming; cutting planes; branch, price and cut.

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" will only work 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 (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 requires the user to 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 contemporaries (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 experimentation.

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 a 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 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. Decomposition for Integer Programming (DIP) (RG05) provides a framework for solving MILP problems using 3 different methods: 1) branch and cut; 2) branch, price and cut; and 3) branch, decompose and cut.[1] In our research we extended PuLP so that a MILP could be defined and solved using DIP within a Python file. We also extended DIP to enable routines defined within the PuLP Python file to be called by the DIP callback functions. In addition to the existing DIP callback functions, we modified DIP to include advanced branching as a callback function. Table 1 gives the mapping of DIP callback functions to Dippy routines. Variable scope within Python means that that these user-defined

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

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

DIP callback function	**Description**	**Dippy mapping**
`chooseBranchSet`	Selects sets of variables and corresponding bounds for the down and up nodes of a branch	`branch_method`
`generateCuts`	Generates user-defined cuts for the current node solution	`generate_cuts`
`APPisUserFeasible`	Checks if the current node solution is feasible, i.e., there are no more user-defined cuts to be generated	`is_solution_feasible`
`APPheuristics`	Runs node heuristics using the current node solution as an input	`heuristics`
`solveRelaxed`	Solves a specified subproblem (relaxation) to generate master problem variables (columns)	`relaxed_solver`
`generateInitVars`	Finds initial master problem variables (columns)	`init_vars`

Table 1: Summary of Dippy callback functions

The rest of this article is structured as follows. Section 2 contains the description and model definition for a case study we will use to demonstrate the effectiveness of Dippy for experimenting with advanced MILP techniques. Next, in sections 3 and 4 we describe how to customise the DIP framework with Dippy. We conclude in section 5 where we discuss how Dippy enhances the ability of researchers to experiment with approaches for solving difficult MILP problems.

2 The Multiple Knapsack Problem (multiknap.py)

Extensions of this problem arise often in MILP problems including network design, rostering and trim loss. The problem is to determine how to place n items into m knapsacks in a way that minimises the wasted space in the knapsacks used. Each item $j = 1, \ldots, n$ has a weight w_j and each knapsack has (weight) capacity W.

The MILP formulation of the multi-knapsack problem is straightforward. The

decision variables are

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is placed in knapsack } i \\ 0 & \text{otherwise} \end{cases}$$

$$y_i = \begin{cases} 1 & \text{if knapsack } i \text{ is used} \\ 0 & \text{otherwise} \end{cases}$$

$$s_i = \text{space left in knapsack } i$$

and the formulation is

$$\begin{aligned} \min \quad & \sum_{i=1}^{m} s_i && \text{(minimise waste)} \\ \text{s.t.} \quad & \sum_{i=1}^{m} x_{ij} = 1, j = 1, \ldots, n && \text{(assignment constraints)} \\ & \sum_{j=1}^{n} w_j x_{ij} + s_i = W y_i, i = 1, \ldots, m && \text{(capacity constraints)} \\ & x_{ij} \in \{0,1\}, y_i \in \{0,1\}, i = 1, \ldots, m, j = 1, \ldots, n \end{aligned}$$

Using PuLP we can easily define this MILP problem in Dippy. The entire input file is given below with summaries for each section.

1. Load PuLP and Dippy;

```
1 import coinor.pulp as pulp
2 from pulp import *
3 import coinor.dippy as dippy
```

2. Get the problem data from another file. This determines $j = 1, \ldots, n$, $i = 1, \ldots, m$, $w_j, j = 1, \ldots, n$ and W;

```
9 from multiknap_ex1 import Weight, ITEMS, KNAPSACKS, Capacity
```

For `multiknap_ex1.py` $n = 5, m = 5, w = (7, 5, 3, 2, 2)^\top$ and $W = 8$.

3. Define the MILP problem and the problem variables;

```
14 prob = dippy.DipProblem("Multi-Knapsack")

16 assign_vars = LpVariable.dicts("InKnapsack",
17                [(i, j) for i in KNAPSACKS
18                        for j in ITEMS],
19                0, 1, LpBinary)
20 use_vars    = LpVariable.dicts("UseKnapsack",
21                KNAPSACKS, 0, 1, LpBinary)
22 waste_vars  = LpVariable.dicts("Waste",
23                KNAPSACKS, 0, Capacity)
```

4. Define the objective function;

```
25 # objective: minimise waste
26 prob += lpSum(waste_vars[i] for i in KNAPSACKS), "min"
```

5. Define the constraints;

```
28 # assignment constraints
29 for j in ITEMS:
30     prob += lpSum(assign_vars[(i, j)] for i in KNAPSACKS) == 1

32 # capacity constraints
33 for i in KNAPSACKS:
34     prob += lpSum(assign_vars[(i, j)] *
35                   Weight[j] for j in ITEMS) + waste_vars[i] == \
36             Capacity * use_vars[i]
```

6. Solve the MILP problem using DIP with default options except for a user-defined tolerance and then display the solution;

```
108 dippy.Solve(prob, {
109     'TolZero': '%s' % tol,
110 })

112 # print solution
113 for i in KNAPSACKS:
114     if use_vars[i].varValue > tol:
115         print "Knapsack ", i, " contains ", \
116               [j for j in ITEMS
117                if assign_vars[(i, j)].varValue > tol]
```

Running the preceding Python code takes 0.94s and 449 nodes and gives the following output:

```
Knapsack  1  contains  [3, 4]
Knapsack  2  contains  [1]
Knapsack  3  contains  [2, 5]
```

Note that DIP uses cuts from the COIN-OR Cut Generator Library (CGL) (LH03) by default. We can implement user-defined cuts in Dippy, but we haven't considered that customisation here.

3 Adding Customised Branching

In DIP we modified `chooseBranchVar` to become `chooseBranchSet`. This makes it possible to define:

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

2. 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 $\lceil\alpha\rceil$ and u.

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

When solving the multiple knapsack problem one difficulty is symmetry in the solution space. Since the knapsacks are identical, solvers consider multiple solutions that differ only in the labelling of the knapsacks. To overcome this constraints for ordering the knapsacks can be ordered:

$$y_i \geq y_{i+1}, i = 1, \ldots, m-1$$

```
39 for n, i in enumerate(KNAPSACKS):
40     if n > 0:
41         prob += use_vars[KNAPSACKS[n-1]] >= use_vars[i]
```

This ordering branch also introduces the opportunity to implement an effective branch on the number of knapsacks. If $\sum_{i=1}^{m} y_i = \alpha \notin \mathbb{Z}$, then:

the branch down restricts	the branch up restricts
$\sum_{i=1}^{m} y_i \leq \lfloor \alpha \rfloor$	$\sum_{i=1}^{m} y_i \geq \lceil \alpha \rceil$
and the ordering means that	and the ordering means that
$y_i = 0, i = \lceil \alpha \rceil, \ldots, m$	$y_i = 0, i = 1, \ldots, \lceil \alpha \rceil$

To implement this branch in Dippy simply requires the definition of the `branch_method`. Note that Dippy can access the variables from the original formulation due to the scope of Python, no complicated indexing or searching is required (also note Python starts its array indexing at 0 – cf. C/C++).

```
44 def choose_antisymmetry_branch(prob, sol):
45     num_knapsacks = sum(sol[use_vars[i]] for i in KNAPSACKS)
46     up = ceil(num_knapsacks)
47     down = floor(num_knapsacks)
48     if (up - num_knapsacks > tol) and \
49        (num_knapsacks - down > tol):    # num_knapsacks is fractional
50         # Define the down branch ubs, lbs = defaults
51         down_branch_ub = [(use_vars[KNAPSACKS[n]], 0) for
52                            n in range(int(down), len(KNAPSACKS))]
53         # Define the up branch lbs, ubs = defaults
54         up_branch_lb = [(use_vars[KNAPSACKS[n]], 1) for
55                         n in range(0, int(up))]

57         return ([], down_branch_ub, up_branch_lb, [])

59 prob.branch_method = choose_antisymmetry_branch
```

The effect of the ordering constraints and the advanced branching are given in section 5.

4 Adding Customised Column Generation

To solve the multiple knapsack problem using branch, price and cut using Dippy, no extra formulation is required. We simply need to identify the constraints

for each subproblem and DIP will automatically generate the Dantzig-Wolfe restricted master problem. To add constraints to a subproblem in Dippy we use the same PuLP syntax, but add them to the appropriate `.relaxation` subproblem. In the multiple knapsack problem we use subproblems to "build" our knapsacks, so we form subproblems from the knapsack capacity constraints:

```
37 # capacity constraints
38 for i in KNAPSACKS:
39     prob.relaxation[i] += lpSum(assign_vars[(i, j)] * Weight[j]
40                                 for j in ITEMS) + waste_vars[i] == \
41                           Capacity * use_vars[i]
```

We can let DIP solve our subproblems, including the generation of initial variables, using its own MILP solver (in our case this is Cbc (LH03)). However, we may be able to speed up the overall solution process by providing our own approaches to solving the pricing subproblems and generating initial variables.

In the pricing subproblem we are looking for knapsacks that provide negative reduced cost. Including an item in the knapsack will add the reduced cost of x_{ij} (`assign_vars[(i, j)]`) but will decrease the waste and hence reduce the contribution of the reduced cost of s_i (`waste_vars[i]`). Here, we calculate the total contribution to reduced cost of adding an item (= reduced cost of $x_{ij}-$ reduced cost of $s_i \times w_j$) and then calculate an items efficiency by dividing this contribution by its weight. Then, we build a minimum reduced cost knapsack by using a greedy algorithm to choose items in order of best efficiency. Since we can access the problem data, variables and their reduced cost, this is straightforward to implement in Dippy:

```
43 def solve_subproblem(prob, index, redCosts, convexDual):
44     knap = index
45
46     # Calculate efficiency of items with negative reduced cost
47     effs = {}
48     for j in ITEMS:
49         effs[j] = (redCosts[assign_vars[(knap, j)]] -
50             redCosts[waste_vars[knap]] * Weight[j]) / Weight[j]
51
52     # Sort the dictionary by value
53     seffs = sorted(effs.items(), key=itemgetter(1))
54
55     # Add efficient items to the knapsack if they fit
56     kp = []
57     waste = Capacity
58     for p in seffs:
59         j = p[0]
60         if Weight[j] <= waste:
61             kp.append(j)
62             waste -= Weight[j]
63
64     # Calculate the reduced cost of the knapsack
65     rc = sum(redCosts[assign_vars[(knap, j)]] for j in kp)
66     rc += redCosts[use_vars[knap]] + redCosts[waste_vars[knap]] * waste
```

```
68      # Return the solution if the reduced cost is low enough
69      if rc < convexDual:
70          var_values = [(assign_vars[(knap, j)], 1) for j in kp]
71          var_values.append((use_vars[knap], 1))
72          var_values.append((waste_vars[knap], waste))

74          dv = dippy.DecompVar(var_values, rc - convexDual, waste)
75          return [dv]

77      return []

79  prob.relaxed_solver = solve_subproblem
```

To generate initial knapsacks we implemented two approaches. The first approach used a first-fit approach and considered the items in order of decreasing weight. The second approach simply placed one item in each knapsack. Using Dippy we can define both approaches at once and then define which one to use by setting the `solve_relaxed` method:

```
81  def first_fit(prob):

83      kps = []
84      sw = sorted(Weight.items(), key=itemgetter(1), reverse=True)
85      unallocated = [s[0] for s in sw]
86      while len(unallocated) > 0:
87          kp = []
88          waste = Capacity
89          j = 0
90          while j < len(unallocated):
91              if Weight[unallocated[j]] <= waste:
92                  item = unallocated[j]
93                  unallocated.remove(item)
94                  kp.append(item)
95                  waste -= Weight[item]
96              else:
97                   j += 1
98          kps.append((kp, waste))

100     bvs = []
101     for k in range(len(kps)):
102         knap = KNAPSACKS[k]
103         kp = kps[k][0]
104         waste = kps[k][1]
105         var_values = [(assign_vars[(knap, j)], 1) for j in kp]
106         var_values.append((use_vars[knap], 1))
107         var_values.append((waste_vars[knap], waste))

109         dv = dippy.DecompVar(var_values, None, waste)
110         bvs.append((knap, dv))

112     return bvs
```

```
114 def one_each(prob):
116     bvs = []
117     for i, knap in enumerate(KNAPSACKS):
118         kp = [ITEMS[i]]
119         waste = Capacity - Weight[ITEMS[i]]
```

⋮ (generate a `dv` and add `bv` as in `first_fit`)

```
127     return bvs
129 prob.init_vars = first_fit
130 ##prob.init_vars = one_each
```

The effects of column generation, included a customised subproblem solver and initial variable generator are shown in section 5.

5 Conclusions

Table 2 shows the effect on the solve time of the ordering constraints (OC), advanced branching (AB), column generation (CG), customised subproblem solver (CS) and initial column generator using first-fit (IC_{FF}) and one item each (IC_{OE}). Note that in some cases the solve process was affected by the order of the items. To account for this variation we ran 5 tests for each strategy with the order of the items randomised. The results in Table 2 show 95% confidence intervals from these tests.

Strategies	Nodes	Time (s)
Default[2]	[502.23, 650.97]	[1.24, 1.51]
OC	[88.80, 96.40]	[0.29, 0.32]
OC + AB	[7, 7]	[0.11, 0.11]
CG	[35.72, 41.48]	[10.38, 13.07]
CG + CS	[49.23, 65.57]	[10.39, 14.44]
CG + IC_{FF}	[26.68, 30.52]	[8.69, 9.36]
CG + IC_{OE}	[26.30, 42.10]	[7.66, 11.58]
CG + CS + IC_{FF}[3]	[9.04, 73.76]	[2.07, 16.76]
CG + CS + IC_{OE}	[58.09, 86.31]	[12.59, 17.66]

Table 2: Results from solving the multiple knapsack problem `multiknap_ex1`

The results in Table 2 show that the ordering constraints and advanced branching significantly reduce the size of the search tree for branch and cut. They also show that the customised subproblem solver with first-fit generation of initial variables may result in a small search tree for branch, price and cut. Note that we did not consider all the possible customisations provided by Dippy, including the ability to generate cuts and add both a root heuristic and node heuristics.

Dippy presents the best of both worlds. It provides a mathematical modelling framework that makes it easy to quickly formulate MILP problems. It also makes it easy to quickly customise the MILP solution framework to experiment with

[2]The DIP default is branch and cut with the CGL library.
[3]The minimum solution scenario had 1 node and took 0.34s.

the effectiveness of advanced MILP techniques for the problem being solved. We have used Dippy successfully to enable final year undergraduate students to experiment with advanced branching, cut generation, column generation and root/node heuristics. Dippy breaks down the barrier to experimentation with advanced MILP approaches for both practitioners and researchers, thus allowing them to concentrate on furthering Operations Research knowledge instead of programming MILP formulations in a low-level language.

6 Acknowledgments

The authors would like to thank the authors of DIP, Ted Ralphs and Matt Galati, for their help throughout this project and the Department of Engineering Science at the University of Auckland for their support of Qi-Shan during this research project.

References

Robin Lougee-Heimer, *The common optimization interface for operations research*, IBM Journal of Research and Development **47** (2003), no. 1, 57–66.

T K Ralphs and M V Galati, *Decomposition in integer programming*, Integer Programming: Theory and Practice (J Karlof, ed.), CRC Press, 2005.