Dippy – a simplified interface for advanced mixed-integer programming

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

February 2011
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, Cameron Walker, Iain Dunning

Department of Engineering Science, The University of Auckland, Auckland, New Zealand

Stuart Mitchell

Stuart Mitchell Consulting, Auckland, New Zealand

Ted Ralphs

Department of Industrial Engineering, Lehigh University, Pennsylvania, USA

April 29, 2014

Abstract

Mathematical modelling languages such as AMPL, GAMS, and Xpress-MP enable mathematical models such as mixed-integer linear programmes (MILPs) to be expressed clearly for solution in solvers such as CPLEX, MINOS and Gurobi. However, some models are sufficiently difficult that they cannot be solved using "out-of-the-box" solvers, and customisation of the solver framework to exploit model-specific structure is required. Many solvers, including CPLEX, Symphony and DIP, enable this customisation by providing "callback functions" that are called at key steps in the solution of a model. This approach traditionally 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 model 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-based mathematical modelling language PuLP and the open source solver DIP. Dippy provides the power of callback functions, but without sacrificing the usability and flexibility of modelling languages. We discuss the link between PuLP and DIP and give examples of how advanced solving techniques can be expressed concisely and intuitively in Dippy.

* Corresponding author.

E-mail address: michael.osullivan@auckland.ac.nz (M. J. O'Sullivan)

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 naturally. Once defined in one of these high-level languages, the MILP can be solved using one of a number of solvers. However these solvers are not effective for all problem instances due to the computational difficulties associated with solving MILPs (an NP-Hard class of problems). Despite steadily increasing computing power and algorithmic improvements for the solution of MILPs in general, in many cases problem-specific techniques need to be included in the solution process to solve problems of a useful size in any reasonable time.

Both commercial solvers – such as CPLEX and Gurobi – and open source solvers – such as CBC, Symphony and DIP (all from the COIN-OR repository [2]) – provide callback functions that allow user-defined routines to be included in the solution framework. To make use of these callback functions the user must first create their MILP problem in a low-level computer programming language (C, C++ or Java for CPLEX; C, C++, C#, Java or Python for Gurobi; C or C++ for CBC, Symphony or DIP). As part of the problem definition, it is necessary to create structures to keep track of the constraints and/or variables. Problem definition in C/C++/Java for a MILP problem of any reasonable size and complexity is a major undertaking and thus a major barrier to the development of customised MILP frameworks by both practitioners and researchers.

Given the difficulty in defining a MILP problem in a low-level language, another alternative for problem formulation is to use a high-level mathematical modelling language. By carefully constructing an indexing scheme, constraints and/or variables in the high-level language can be identified in the low-level callback functions. However implementing the indexing scheme can be as difficult as using the low-level language to define the problem in the first place and does little to remove the barrier to solution development.

The purpose of the research presented here is to demonstrate a tool, Dippy, that supports easy experimentation with and customisation of advanced MILP solution frameworks. To achieve this aim we needed to:

1. provide a modern high-level modelling system that enables users to quickly and easily describe their MILP problems;

2. enable simple identification of constraints and variables in user-defined routines within the solution framework.

The first requirement is satisfied by the modelling language PuLP [3]. Dippy extends PuLP to use the Decomposition for Integer Programming (DIP) solver, and enables user-defined routines, implemented using Python and PuLP, to be accessed by the DIP callback functions. This approach enables constraints or variables defined in the MILP model to be easily accessed using PuLP in the user-defined routines. In addition to this, 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 – it hides the implementation of the master problem and subproblems. This makes it very easy to switch between the two approaches when experimenting with solution methods. All this functionality combines to overcome the barrier described previously and provides researchers, practitioners and students with a simple and integrated way of describing problems and customising the 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, followed by a guide of how to get started with Dippy in section 3. Then, section 4 contains descriptions and model definitions of the case studies we will use to demonstrate the effectiveness of Dippy. In section 5 we describe how Dippy enables experimentation with advanced techniques within DIP's MILP solution framework to improve solution times. We demonstrate these techniques using example code for the case studies from section 4. We conclude in section 6 where we discuss how this project enhances the ability of researchers to experiment with approaches for solving difficult MILP problems. We also demonstrate that DIP (via PuLP and Dippy) is competitive with leading commercial (Gurobi) and open source (CBC) solvers.

2 Combining DIP and PuLP

Dippy is the primarily the "glue" between two different technologies: PuLP and DIP.

PuLP [3] is a mathematical modelling language and toolkit that uses Python. Users can define MILP problems and solve them using a variety of solvers including CPLEX, Gurobi and CBC. PuLP's solver interface is modular and thus can be easily extended to use other solvers such as DIP. For more details on PuLP see the PuLP project in the COIN-OR repository [2].

Decomposition for Integer Programming (DIP) [5] provides a framework for solving MILP problems using 3 different methods[1]:

1. "branch-and-cut",
2. "branch-price-and-cut",
3. "decompose-and-cut".

In this paper we will restrict our attention to branch-and-cut and branch-price-and-cut.

Branch-and-cut uses the classic branch-and-bound approach for solving MILPs combined with the cutting plane method for removing fractionality encountered at the branch-and-bound nodes. This framework is the basis of many state-of-the-art MILP solvers including Gurobi and CBC. DIP provides callback functions that allow users to customise the solution process by adding their own cuts and running heuristics at each node.

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

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

The cut generation and heuristic callback functions mentioned previously can also be used for branch-price-and-cut. 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 [5].

In addition to the DIP callback functions (see §2.1), we modified DIP to add 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 replaced `chooseBranchVar` in the DIP source with a new function `chooseBranchSet`. This is a significant change to branching in DIP that 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 §5.1. From DIP, `chooseBranchSet` calls `branch_method` in Dippy.

Customised Cuts We modified `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 modified the DIP function `solveRelaxed` to call `relaxed_solver` in Dippy. This enables the user to utilise the master problem dual variables to produce solutions to subproblems (and so add columns to 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 modified `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.

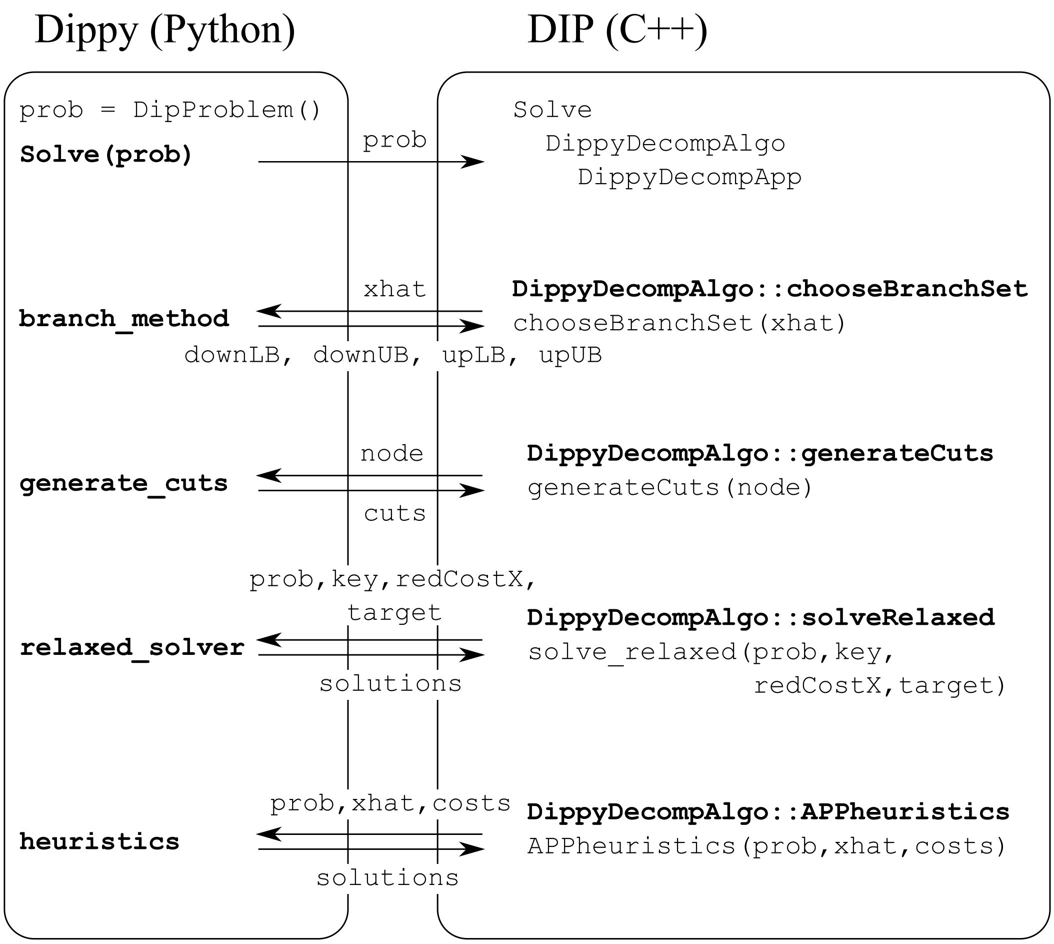

Figure 1: Key components of interface between Dippy and DIP.

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++. DIP `Solve` creates a `DippyDecompAlgo` object that contains a `DippyDecompApp` object, both of which are populated by data from `prob`. As DIP `Solve` proceeds branches are created by the `DippyDecompAlgo` object using `chooseBranchSet` which passes the current node's fractional solution `xhat` back to the `branch_method` function in the `DipProblem` object `prob`. 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 `node` to the `DipProblem` object's `generate_cuts` function. This function generates any customised cuts and returns a list, `cuts`, back to `DippyDecompApp::generateCuts`. These interfaces are replicated for the other callback functions provided by Dippy.

3 Getting Started with Dippy

*** Put citation info for DIP and Dippy here, make it easy to get 4 citations: DIP paper, DIP software, Dippy paper (this one until journal article is "born" and Dippy software ***

3.1 Installing Dippy

3.2 Visualising Search Trees

4 Case Studies

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

1. the bin packing 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.

4.1 The Bin Packing Problem (bin_pack_func.py)

The solution of the bin packing problem determines where, amongst m "bins", to place n "items" of various "sizes" in a way that (in this case study) minimises the wasted "capacity" of the bins. Each product $j = 1, \ldots, n$ has a size s_j and each bin has capacity C. Extensions of this problem arise often in MILP in problems including network design and rostering.

The MILP formulation of the bin packing problem is straightforward. The decision variables are

$$
\begin{aligned}
x_{ij} &= \begin{cases} 1 & \text{if item } j \text{ is placed in bin } i \\ 0 & \text{otherwise} \end{cases} \\
y_i &= \begin{cases} 1 & \text{if a facility is located at location } i \\ 0 & \text{otherwise} \end{cases} \\
w_i &= \text{“wasted” capacity at location } i
\end{aligned}
$$

and the formulation is

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{m} w_i \\
\text{s.t.} \quad & \sum_{i=1}^{m} x_{ij} = 1, j = 1, \ldots, n && \text{(each product produced)} \\
& \sum_{j=1}^{n} r_j x_{ij} + w_i = C y_i, i = 1, \ldots, m && \text{(aggregate capacity at location } i\text{)} \\
& x_{ij} \leq y_i, i = 1, \ldots, m, j = 1, \ldots, n && \text{(disaggregate capacity at location } i\text{)} \\
& x_{ij} \in \{0, 1\}, w_i \geq 0, y_i \in \{0, 1\}, i = 1, \ldots, m, j = 1, \ldots, n
\end{aligned}
$$

Note that the disaggregate capacity constraints are not necessary for defining the solution, but tighten the MILP formulation (i.e., remove factional solutions from the solution space). Using PuLP we can easily define this MILP problemm in Dippy. The entire input file is given below with a summary for each fragment.

1. Load PuLP and Dippy;

```python
Bin_antisymmetry = False
Item_antisymmetry = False

Symmetry_branch = False
Most_use_branch = False
Most_assign_branch = False

import sys

# Import classes and functions from PuLP
from pulp import LpVariable, lpSum, LpBinary, LpStatusOptimal

# Import any customised paths
try:
    import path
except ImportError:
    pass
```

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

```python
import sys
```

For facility_ex1.py $n = 5, m = 5, r = (7, 5, 3, 2, 2)^\top$ and $C = 8$.

3. Define the MILP problem and the problem variables;

```python
# Import any customised paths
try:
    import path
except ImportError:
    pass

# Import dippy (local copy first,
# then a development copy - if python setup.py develop used,
# then the coinor.dippy package
try:
```

4. Define the objective function;

```python
except ImportError:
    try:
```

5. Define the constraints;

```python
    except ImportError:
        import coinor.dippy as dippy

from math import floor, ceil

class BinPackProb:
    def __init__(self, ITEMS, volume, capacity):
        self.ITEMS = ITEMS
        self.volume = volume
        self.BINS = range(len(ITEMS)) # Create 1 bin for each item,
                                      # indices start at 0
        self.capacity = capacity

def formulate(bpp):
```

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

```python
def fit(prob, order):
    bpp         = prob.bpp
    use_vars    = prob.use_vars
    assign_vars = prob.assign_vars
    waste_vars  = prob.waste_vars
    tol         = prob.tol

    sol = {}
```

Running the preceding Python codes takes 1.17s of CPU time, creates a tree with 375 nodes and gives the following output

```
Location  1  produces  [1]
Location  4  produces  [4, 5]
Location  5  produces  [2, 3]
```

Note that DIP uses cuts from the Cut Generator Library (CGL) [2] by default. We can turn all cuts off by setting the `generateCuts` flag to 0 and turn CGL cuts off by setting the `CutCGL` flag to 0. We will use these settings to explore the effect of user-defined cuts in section **??**.

We will use the Bin Packing Problem to demonstrate the implementation of customised branching rules, custom cuts, heuristics, and a column generation algorithm.

The solution of the problem determines which, of m bins, to use and also places n items of various sizes into the bins in a way that (in this version) minimises the wasted capacity of the bins. Each item $j = 1, \ldots, n$ has a size s_j and each bin has capacity C. Extensions of this problem arise often in MILP in problems including network design and rostering.

The MILP formulation of the bin packing problem is straightforward. The decision variables are

$$
\begin{aligned}
x_{ij} &= \begin{cases} 1 & \text{if item } j \text{ is placed in bin } i \\ 0 & \text{otherwise} \end{cases} \\
y_i &= \begin{cases} 1 & \text{if bin } i \text{ is used} \\ 0 & \text{otherwise} \end{cases} \\
w_i &= \text{“wasted” capacity in bin } i
\end{aligned}
$$

and the formulation is

$$
\begin{array}{lll}
\min & \displaystyle\sum_{i=1}^{m} w_i & \\
\text{s.t.} & \displaystyle\sum_{i=1}^{m} x_{ij} = 1, j = 1, \ldots, n & \text{(each item packed)} \\
& \displaystyle\sum_{j=1}^{n} s_j x_{ij} + w_i = C y_i, i = 1, \ldots, m & \text{(aggregate packing for bin } i\text{)} \\
& x_{ij} \leq y_i, i = 1, \ldots, m, j = 1, \ldots, n & \text{(individual packing for bin } i\text{)} \\
& x_{ij} \in \{0, 1\}, w_i \geq 0, y_i \in \{0, 1\}, i = 1, \ldots, m, j = 1, \ldots, n &
\end{array}
$$

Note that the constraints for the individual packing in a bin are not necessary for defining the solution, but tighten the MILP formulation by removing fractional solutions from the solution space. Before looking at the advanced techniques that can be easily implemented using Dippy, we will examine how to formulate the bin packing problem in PuLP and Dippy.

4.2 Formulating the Bin Packing Problem

Before formulating we need to include the PuLP and Dippy modules into Python

```
Bin_antisymmetry = False
Item_antisymmetry = False

Symmetry_branch = False
Most_use_branch = False
Most_assign_branch = False

import sys

# Import classes and functions from PuLP
from pulp import LpVariable, lpSum, LpBinary, LpStatusOptimal

# Import any customised paths
try:
    import path
except ImportError:
    pass

# Import dippy (local copy first,
```

and define a class to hold a bin packing problem's data

```
    import dippy
except ImportError:
    try:
        import src.dippy as dippy
    except ImportError:
        import coinor.dippy as dippy
```

The `formulate` function is defined with a bin packing problem object as input and creates a `DipProblem` (with some display options defined)

```

class BinPackProb:
    def __init__(self, ITEMS, volume, capacity):
        self.ITEMS = ITEMS
        self.volume = volume
        self.BINS = range(len(ITEMS)) # Create 1 bin for each item,
```

Then, using the bin packing problem object's data (i.e., the data defined within `bpp`), the decision variables

```
        self.capacity = capacity

def formulate(bpp):
    prob = dippy.DipProblem("Bin Packing",
                            display_mode = 'xdot',
#                           layout = 'bak',
```

objective function

```
                            )
```

and constraints are defined

```
    assign_vars = LpVariable.dicts("x",
                                   [(i, j) for i in bpp.ITEMS
                                    for j in bpp.BINS],
                                   cat=LpBinary)
    use_vars    = LpVariable.dicts("y", bpp.BINS, cat=LpBinary)
    waste_vars  = LpVariable.dicts("w", bpp.BINS, 0, None)

    prob += lpSum(waste_vars[j] for j in bpp.BINS), "min_waste"

    for j in bpp.BINS:
        prob += lpSum(bpp.volume[i] * assign_vars[i, j]
```

Finally, the bin packing problem object and the decision variables are all "embedded" within the DipProblem object, prob, and this object is returned (note that the objective function and constraints could also be similarly embedded)

```
        prob += lpSum(assign_vars[i, j] for j in bpp.BINS) == 1

    for i in bpp.ITEMS:
        for j in bpp.BINS:
            prob += assign_vars[i, j] <= use_vars[j]

    if Bin_antisymmetry:
        for m in range(0, len(bpp.BINS) - 1):
```

In order to solve the bin packing problem, only the DipProblem object, prob, is required (note that no dippyOpts are specified, so the Dippy defaults are used)

```
            bounds = most_frac_use(prob, sol)
def my_heuristics(prob, xhat, cost):
            sol = first_fit(prob)
    else:
        if prob.node_heuristic:
            sol = frac_fit(prob, xhat)

    if sol is not None:
        return [sol]
```

To solve an instance of the bin packing problem, the data needs to be specified and then the problem formulated and solved

```
from bin_pack_func import BinPackProb, formulate, solve

if __name__ == '__main__':
    # Python starts here
    bpp = BinPackProb(ITEMS  = [1, 2, 3, 4, 5],
                      volume = {1: 2, 2: 5, 3: 3, 4: 7, 5: 2},
                      capacity = 8)

    prob = formulate(bpp)
```

```
    # Set a zero tolerance (Mike Saunders' "magic number")
    prob.tol = pow(pow(2, -24), 2.0 / 3.0)
```

```python
    xopt = solve(prob)

    if xopt is not None:
        for var in prob.variables():
            print var.name, "=", xopt[var]
    else:
        print "Dippy could not find and optimal solution"
```

Solving this bin packing problem instance in Dippy gives the branch-and-bound tree shown in figure 2 (note thet the integer solution found – indicated in blue `S: 5.0` – bounds all other nodes in the tree) with the final solution packing items 1 and 2 into bin 0 (for a waste of 1), items 3 and 5 into bin 1 (for a waste of 3) and item 4 into bin 3 (for a waste of 1).

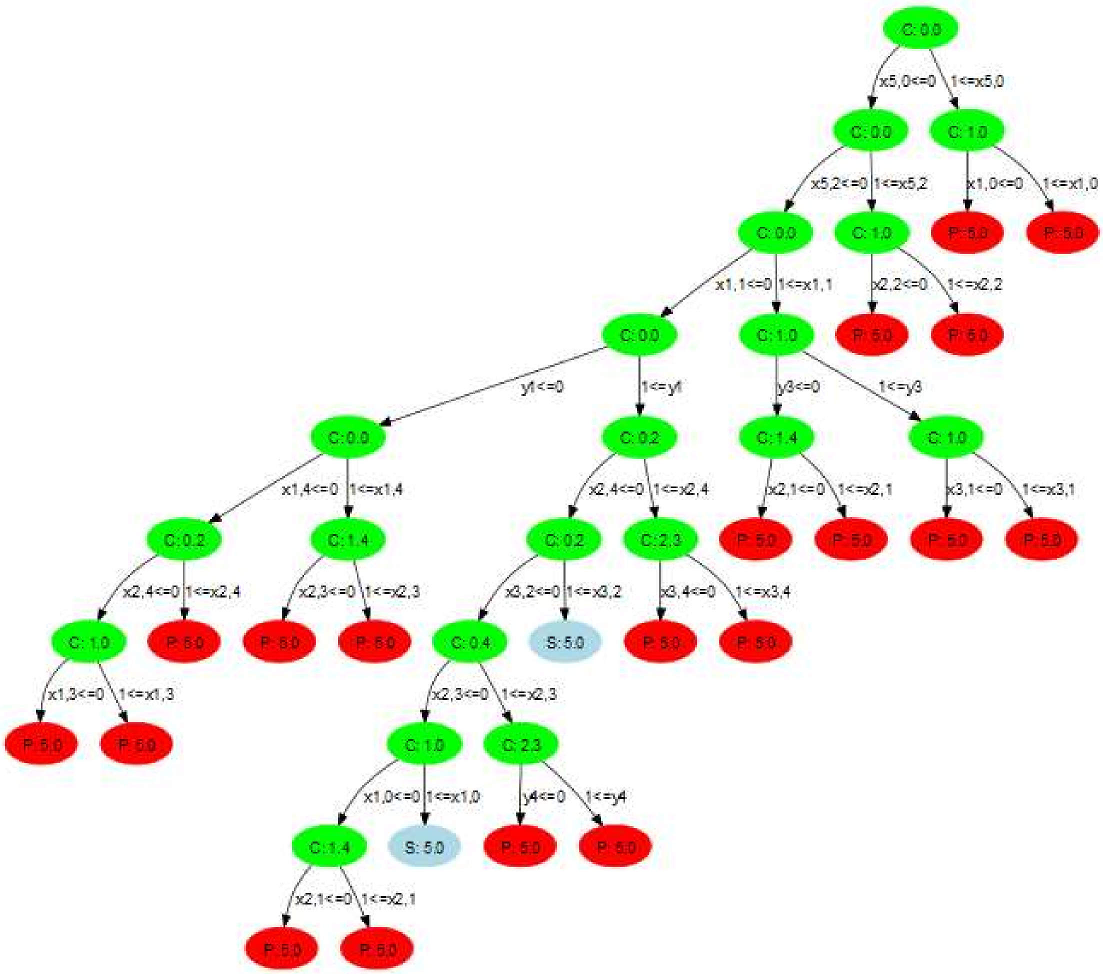

Figure 2: Branch-and-bound tree for bin packing problem instance.

4.3 The Coke Supply Chain Problem (coke.py)

This case study is sourced from the Operations Research Web in the Department of Engineering Science TWiki [7] (and was originally adapted from Leyland et al. [1]). There are 6 coal mines that produce coal. The coal is transported from the 6 mines to a coke-making plant where it is converted to coke using "thermal decomposition". Every tonne of coke produced by thermal decomposition requires 1.3 tonnes of coal. From the coke-making plants the coke is transported to one of 6 customers. There are 6 locations where coke-making plants can be constructed. There are 6 different size plants that can be constructed at each location.

The size of a plant determines the coke processing level in kilotonnes/year the plant can produce. Table 1 shows the different plant sizes with their corresponding processing levels and construction cost in million RMB.

Plant Size	Processing Level (kT/year)	Cost (MRMB)
1	75	4.4
2	150	7.4
3	225	10.5
4	300	13.5
5	375	16.5
6	450	19.6

Table 1: Possible plant sizes

To get this problem into Dippy we use the PuLP modelling language. The entire input file is given below with a summary for each fragment.

1. Load PuLP and Dippy;

```
BIG_M = 1e10
```

2. Define the coke-from-coal conversion rate and a big-M variable for use in later constraints;

```
    "M1": 25.8,
    "M2": 728,
```

3. Define the supply of coal at the mines, the possible locations and construction costs of the coke-making plants and the demand for coke from the customers.

```
    "M4": 49,
    "M5": 36.9,
    "M6": 1100,
}
MINES = MINE_SUPPLY.keys()
MINES.sort()
```

```
LOCATIONS = ["L1", "L2", "L3", "L4", "L5", "L6"]

SIZE_COSTS = {
    0: 0,
    75: 4.4,
```

```
    225: 10.5,
    300: 13.5,
    375: 16.5,
    450: 19.6,
}
SIZES = SIZE_COSTS.keys()
SIZES.sort()

CUSTOMER_DEMAND = {
    "C1": 83,
    "C2": 5.5,
    "C3": 6.975,
    "C4": 5.5,
    "C5": 720.75,
    "C6": 5.5,
}
CUSTOMERS = CUSTOMER_DEMAND.keys()
CUSTOMERS.sort()

MINE_TRANS_DATA = """
        L1      L2      L3      L4      L5      L6
M1      231737  46813   79337   195845  103445  45186
```

4. Define the transportation costs from the mines to the coke-making plants and the coke-making plants to the customers in two tables (we also define a function `read_table` to read these tables, but this function is omitted for brevity);

```
M3      45170   93159   156241  218655  103802  119616
M4      149925  254305  76423   123534  151784  104081
M5      152301  205126  24321   66187   195559  88979
M6      223934  132391  51004   122329  222927  54357
"""

CUST_TRANS_DATA = """
        L1      L2      L3      L4      L5      L6
C1      6736    42658   70414   45170   184679  111569
C2      217266  227190  249640  203029  153531  117487
C3      35936   28768   126316  2498    130317  74034
C4      73446   52077   108368  75011   49827   62850
C5      174664  177461  151589  153300  59916   135162
C6      186302  189099  147026  164938  149836  286307
"""

def read_table(data, coerce, transpose=False):
    lines = data.splitlines()
    headings = lines[1].split()
```

5. Read the data from the tables;

```
CUST_TRANS = read_table(CUST_TRANS_DATA, int, \
                        transpose=True)
```

6. Define the arcs and their costs from the mine → plant and plant → customer costs;

```
ARC_COSTS.update(CUST_TRANS)
ARCS = ARC_COSTS.keys()

def cross(i1, i2):
```

7. Define a 2-dimensional set for the plant sizes at each location;

```
    for a in i1:
        for b in i2:
```

8. Create a `DipProblem` (extended from `LpProblem` in PuLP). Add binary variables that determine the plant sizes at each location and (non-negative) integer variables that determine the flow (in coal from the mines to the plants and coke from the plants to the customers) transported through the network;

```
    return r

LOC_SIZES = cross(LOCATIONS, SIZES)

prob = dippy.DipProblem("Coke", LpMinimize)

# create variables
buildVars = LpVariable.dicts("Build", LOC_SIZES, None, \
                             None, LpBinary)
```

9. Add the objective of minimising total cost = building costs (converted from MRMB to RMB) + transportation costs;

```

# create arcs
flowVars = LpVariable.dicts("Arcs", ARCS)
for a in ARCS:
    flowVars[a].bounds(0, BIG_M)
```

10. Add constraints that limit the flow of coke out of a coke-making plant depending on the capacity plant constructed;

```

# objective
prob += 1e6 * lpSum(buildVars[(l, s)] * SIZE_COSTS[s] \
                    for (l, s) in LOC_SIZES) + \
              lpSum(flowVars[(s, d)] * ARC_COSTS[(s, d)] \
                    for (s, d) in ARCS), "min"
```

11. Add constraints that limit the number of coke-making plants built at any single location to be one (Note. there is a size with capacity 0 if no plant will be built);

```
# plant availability
for loc in LOCATIONS:
    prob += lpSum(flowVars[(loc, i)] for i in CUSTOMERS) \
            <= lpSum(buildVars[(loc, s)] *s for s in SIZES)
```

12. Add constraints to conserve flow at the mines ($\leq$ supply), coke-making plants (flow in = coke-from-coal conversion rate $\times$ flow out) and customers ($\geq$ demand);

```
# one size
for loc in LOCATIONS:
    prob += lpSum(buildVars[(loc, s)] for s in SIZES) == 1

# conserve flow (mines)
# flows are in terms of tonnes of coke
for m in MINES:
    prob += lpSum(flowVars[(m, j)] for j in LOCATIONS) <= \
            MINE_SUPPLY[m]/CC

for loc in LOCATIONS:
    prob += lpSum(flowVars[(m, loc)] for m in MINES) - \
            lpSum(flowVars[(loc, c)] for c in CUSTOMERS) \
            >= 0
```

13. Solve the MILP problem using DIP and display the solution in tabular form;

```
                                             for j in range(0, i)]

                return ([], down_branch_ub, [], up_branch_ub)

prob.branch_method = do_branch

dippy.Solve(prob, {
    'CutCGL': '0',
})

def print_table(rows, cols, fn):
    print "\t", "\t".join(cols)
    for r in rows:
        print r,"\t", "\t".join(str(fn(r,c)) for c in cols)

def print_var_table(rows, cols, var, fn=lambda x: x):
    print_table(rows, cols, lambda x, y:
                fn(var[(x,y)].varValue))
```

The preceding Python code defines and solves the Coke Supply Chain Problem. The solution takes 1.09s of CPU time and creates a tree using 201 nodes. The output defines plants to be built at locations 1, 5 and 6 and also defines shipments of coal and coke between the mines, plants and customers:

```
Build L1 150 (1.0)
```

```
Build L2 0 (1.0)
Build L3 0 (1.0)
Build L4 0 (1.0)
Build L5 450 (1.0)
Build L6 300 (1.0)

        L1       L2       L3       L4       L5       L6
M1      0.0      0.0      0.0      0.0      0.0      25.8
M2      0.0      0.0      0.0      0.0      0.0      340.475
M3      124.1175 0.0      0.0      0.0      585.0    0.0
M4      0.0      0.0      0.0      0.0      0.0      0.0
M5      0.0      0.0      0.0      0.0      0.0      0.0
M6      0.0      0.0      0.0      0.0      0.0      0.0

        C1       C2       C3       C4       C5       C6
L1      83.0     0.0      6.975    0.0      0.0      5.5
L2      0.0      0.0      0.0      0.0      0.0      0.0
L3      0.0      0.0      0.0      0.0      0.0      0.0
L4      0.0      0.0      0.0      0.0      0.0      0.0
L5      0.0      0.0      0.0      0.0      450.0    0.0
L6      0.0      5.5      0.0      5.5      270.75   0.0
```

4.4 The Travelling Salesperson Problem (tsp.py)

This case study is a small travelling salesperson (TSP) example. This problem differs from the previous case studies (§**??** and §4.3) in that it can't be expressed explicitly for any reasonable size problem. To completely define the travelling salesperson (TSP) problem requires a number of subtour elimination constraints that is $O(2^n)$ where $n = |N|$ is the number of locations the salesperson must visit in their tour. The standard way to solve TSP problems is to use a formulation without any subtour elimination constraints and dynamically add only the subtour elimination constraints needed to define an optimal tour. Here we will use PuLP to define the MILP formulation without subtour elimination constraints.

1. Load PuLP, Dippy and the square root function from the math module;

```python
from pulp import *
import dippy
from math import sqrt
```

2. Define the cities and their locations in the xy-plane. Also, define empty structures for arcs between each pair of cities and int/out of cities;

```
# x,y coords of cities
CITY_LOCS = [(0, 2), (0, 4), (1, 2), (1, 4), \
             (4, 1), (4, 4), (4, 5),  (5, 0), \
             (5, 2), (5, 5)]
CITIES = range(len(CITY_LOCS))

ARCS = [] # list of arcs (no duplicates)
ARC_COSTS = {} # distance

# for each city, list of arcs into/out of
CITY_ARCS = [[] for i in CITIES]
```

3. Define the Euclidean distance using `sqrt`;

```
# use 2d euclidean distance
def dist(x1, y1, x2, y2):
    return sqrt((x1-x2)**2 + (y1-y2)**2)
```

4. Define the arcs between cities, the arcs in/out of a city and the cost of the arcs as the distance between cities;

```
# construct list of arcs
for i in CITIES:
    i_x, i_y = CITY_LOCS[i]
    for j in CITIES[i+1:]:
        j_x, j_y = CITY_LOCS[j]
        ARC_COSTS[(i,j)] = dist(i_x, i_y, j_x, j_y)
        ARCS.append((i, j))
        CITY_ARCS[i].append((i, j))
        CITY_ARCS[j].append((i, j))
```

5. Use the standard TSP MILP formulation without any subtour constraints. The standard formulation is:

$$
\begin{aligned}
\min \sum_{(i,j)\in A} & c_{ij}x_{ij} \\
\sum_{\substack{(i,j)\in A \\ i=k \text{ or } j=k}} & x_{ij} = 2, k \in N.
\end{aligned}
$$

```
prob = dippy.DipProblem()

arc_vars = LpVariable.dicts("UseArc", ARCS, 0, 1, LpBinary)

# objective
prob += lpSum(ARC_COSTS[x] * arc_vars[x] for x in ARCS)
```

```
# degree constraints
for city in CITIES:
    prob += lpSum(arc_vars[x] for x in CITY_ARCS[city]) \
```

6. Solve the TSP using DIP and display the minimum cost tour;

```
```

Solving the TSP using DIP takes 0.13s of CPU time and gives the following solution:

```
(5, 9) 1.0
(4, 7) 1.0
(1, 3) 1.0
(4, 8) 1.0
(5, 6) 1.0
(6, 9) 1.0
(2, 3) 1.0
(0, 1) 1.0
(7, 8) 1.0
(0, 2) 1.0
```

with 3 subtours:

1. $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0$;
2. $4 \rightarrow 7 \rightarrow 8 \rightarrow 4$;
3. $5 \rightarrow 6 \rightarrow 9 \rightarrow 5$.

The optimal TSP solution can only be found by adding user-defined cuts that remove subtours. Section **??** describes how to implement these user-defined cuts in Dippy and shows how these cuts combine with the CGL cuts to efficiently solve this TSP.

4.5 The Cutting Stock Problem (cutting_stock.py)

This case study also come from the Operations Research Web in the Department of Engineering Science TWiki [4]. The solution of this problem defines cutting patterns to produce the required demand for items from standard items. In this case study the demand is for variable length sponge rolls to be cut from standard length rolls. The entire input file is given below with a summary for each fragment.

1. Load PuLP and Dippy;

```
from pulp import *
import dippy
```

2. Define the length of sponge rolls required and the demand for each length of sponge roll (note, some variations of demand are shown but have been commented out);

```
length = {
"9cm": 9,
"7cm": 7,
"5cm": 5
}
```

```
ITEMS = length.keys()

demand = {
"9cm": 3,
"7cm": 2,
"5cm": 2
}

total_patterns = sum(demand[i] for i in ITEMS)

total_length = {}
for p in range(total_patterns):
    total_length[p] = 20
```

3. Define the maximum number of possible patterns used for cutting the standard rolls (at most one standard roll for each sponge roll needed) and the length of the standard rolls;

```
def cross(i1, i2):
    r = []
    for a in i1:
        for b in i2:
            r.append((a, b))
```

4. Define a two dimensional set of items cut from patterns (cf. 7 from §4.3);

```
CUTS = cross(PATTERNS, ITEMS)
```

5. Create a `DipProblem`. Add binary variables that determine if each pattern is used and (non-negative, bounded) integer variables that define the number of sponge rolls of each length cut from a particular pattern.

```
34 prob = dippy.DipProblem("Python", LpMinimize)

36 # create variables

38 useVars = LpVariable.dicts("Use", PATTERNS, 0, 1, LpBinary)
39 prob.useVars = useVars
```

Note that normally we would define an integer variable that defines how many times a pattern is used and, thus, need less patterns. However, DIP does not (yet) solve identical subproblems simultaneously, so we need one subproblem for each pattern cut;

6. We want to minimise the total number of standard rolls used;

```
42 prob.cutVars = cutVars
```

7. We want to meet demand for sponge rolls;

```
45 prob += lpSum(useVars[p] for p in PATTERNS), "min"

47 # Meet demand
48 for i in ITEMS:
```

8. Add constraints that make sure patterns are used "in order" (these constraints are not strictly necessary but remove symmetry in the solution space);

```
50             >= demand[i]

52 # Ordering patterns
53 for i, p in enumerate(PATTERNS):
```

9. Create one subproblem for each pattern that makes sure the sponge rolls cut from the standard roll in the pattern do not exceed the length of the standard roll. Note the `relaxation[p]` on line 57. This adds the constraint to the Dantzig-Wolfe subproblem if branch, price and cut is used (for more details see section **??**);

```
55          prob += useVars[p] >= useVars[PATTERNS[i+1]]

57 for p in PATTERNS:
58     prob.relaxation[p] += \
59     lpSum(length[i] * cutVars[(p, i)] for i in ITEMS) \
```

10. Solve the Sponge Roll Production Problem using branch, price and cut. Display the patterns used and the sponge rolls cut from those patterns. Note that the `doPriceCut` options is turned on (set to 1). This means that DIP will use branch, price and cut instead of branch and cut;

This problem takes 33.31s of CPU time and requires 175 nodes in the branch-and-bound tree for the master problem. The solution uses 2 standard rolls cut as follows:

- Standard roll 0: 2 $\times$ 5cm rolls and 1 $\times$ 9cm roll = 19cm used (1cm wasted);
- Standard roll 1: 2 $\times$ 5cm rolls and 1 $\times$ 7cm roll = 17cm used (3cm wasted).

4.6 The Wedding Planner Problem (wedding.py)

This case study is taken from the PuLP documentation [3]. Given a list of wedding attendees, a wedding planner must come up with a seating plan to minimise the unhappiness of all of the guests. The unhappiness of guest is defined as their maximum unhappiness at being seated with each of the other guests at their table, i.e., it is a pairwise function. The unhappiness of a table is the maximum unhappiness of all the guests at the table. All guests must be seated and there is a limited number of seats at each table.

The wedding planner problem is a set partitioning problem. The set of guests G must be partitioned into multiple subsets, with each subset seated at the same table. The cardinality of the subsets is determined by the number of seats at a table and the unhappiness of a table can be determined by the subset. The MILP formulation is:

$$
\begin{aligned}
x_{gt} &= \begin{cases} 1 & \text{if guest } g \text{ sits at table } t \\ 0 & \text{otherwise} \end{cases} \\
u_t &= \text{unhappiness of table } t \\
S &= \text{number of seats at a table} \\
U(g,h) &= \text{unhappiness of guests } g \text{ and } h \text{ if they are seated at the same table}
\end{aligned}
$$

$$
\begin{aligned}
\min \quad \sum_{t \in T} u_t & \quad \text{(total unhappiness of the tables)} \\
\sum_{g \in G} x_{gt} & \leq S, t \in T \\
\sum_{t \in T} x_{gt} & = 1, g \in G \\
u_t & \geq U(g,h)(x_{gt} + x_{ht} - 1), t \in T, g < h \in G
\end{aligned}
$$

To get this problem into Dippy we use the PuLP modelling language. The entire model follows with a summary for each fragment:

1. Load PuLP and Dippy;

```
try:
    import path
```

2. Define the unhappiness function for the guests (in this case we use letters in the alphabet as guests and the "distance" between two letters in the guest list as their unhappiness at being seated together);

```
except ImportError:
    try:
        import src.dippy as dippy
    except ImportError:
        import coinor.dippy as dippy
```

3. Get the problem data from an external program (this is used to test various inputs to the MILP formulation);

```
max_table_size = 4
```

4. Create a the `DipProblem` for Dippy;

```
    """
    Return the happiness (0 is the best) of allocating two
```

5. Create a set for the tables and also for all possible seatings, i.e., pairs $g \in G, t \in T$;

```
    """
    return abs(ord(guest_a) - ord(guest_b))

#create the set of possible tables
tables = range(max_tables)
```

6. Create the seating variables $x_{gt}, g \in G, t \in T$;

```
                            for t in tables]

#create a binary variable to model if a guest sits at a particular table
x = pulp.LpVariable.dicts('possible_seatings', possible_seatings,
                            lowBound = 0,
                            upBound = 1,
                            cat = pulp.LpInteger)
```

7. Create the table unhappiness variables $u_t, t \in T$;

```
seating_model = dippy.DipProblem("Wedding Seating Model (DIP)", pulp.LpMinimiz
                                 display_mode = 'xdot', display_interval = 0)

#specify the maximum number of guests per table
for table in tables:
```

8. Create the objective that minimises the total unhappiness of the tables;

```
                                            for guest in guests]) <= \
                                            max_table_size, \
```

9. Create the constraints for: 1) the number of seats at a table; 2) ensuring each guest is seated; and 3) defining table unhappiness;

```

#A guest must seated at one and only one table
for guest in guests:
    seating_model += (sum([x[(guest, table)] for table in tables]) == 1,
                      "Must_seat_%s"%guest)
```

```
#create a set of variables to model the objective function
possible_pairs = [(a, b) for a in guests for b in guests if ord(a) < ord(b)]
happy = pulp.LpVariable.dicts('table_happiness', tables,
                              lowBound = 0,
                              upBound = None,
                              cat = pulp.LpContinuous)

seating_model += sum([happy[table] for table in tables])

#create constraints for each possible pair
for table in tables:
    for (a, b) in possible_pairs:
        seating_model.relaxation[table] += \
            happy[table] >= (happiness(a, b) * (x[(a, table)] +
                                                 x[(b, table)] - 1))

def relaxed_solver(prob, table, redCosts):
```

10. Solve the problem using branch, price and cut;

Note the `relaxation[table]` syntax on lines 55 and 72. This defines a separate subproblem for each table that contains the constraint for the number of seats at a table and the constraints defining table unhappiness. These table subproblems are used in branch, price and cut.

For a simple example, where the wedding guests are { A, B, C, D, E, F, G, H, I, J, K }, the solution time is 1.28s of CPU time and the tree consists of 1395 nodes. The solution is

```
Table 0 = ['D', 'E', 'F', 'G']
Table 1 = ['A', 'B', 'C']
Table 2 = ['H', 'I', 'J', 'K']
```

5 Dippy in Practice

5.1 Adding Customised Branching

In §2.1 we explained the modifications made to DIP and how a simple variable branch would be implemented. The DIP function `chooseBranchSet` calls Dippy's `branch_method` at fractional nodes. The function `branch_method` has two inputs supplied by DIP:

1. `prob` – the `DipProblem` being solved;

2. `sol` – an indexable object representing the solution at the current node.

We define `branch_method` using these inputs and the same PuLP structures used to defined the model, allowing Dippy to access the variables from the original formulation and eliminating any need for complicated indexing.

We can explore custom branching rules that leverage constraints to reduce the symmetry in the solution space of the bin packing problem. Inefficiencies arise from solvers considering multiple equivalent solutions that have identical objective function values and differ only in the subset of the identical bins used. One way to address this is to add a constraint that determines the order in which the bins can be considered:

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

```
                  == bpp.capacity * use_vars[j]
```

This change results in a smaller branch-and-bound tree (see figure 3) that provides the same solution but with bin 0 used in place of bin 3, i.e., a symmetric solution, but with the bins now used "in order".

These ordering constraints also introduce the opportunity to implement an effective branch on the number of facilities:

If $\sum_{i=1}^{m} y_i = \alpha \notin \mathbb{Z}$, then:

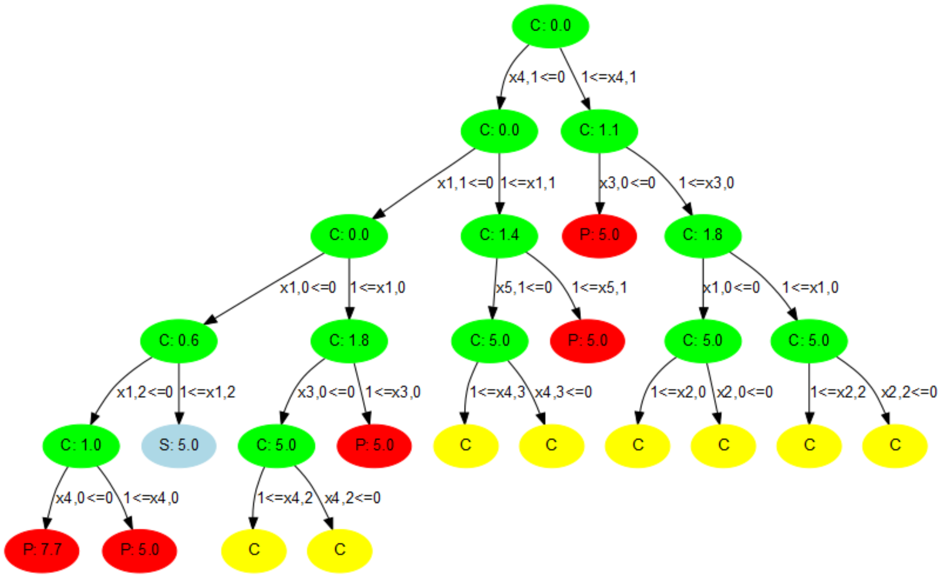

Figure 3: Branch-and-bound tree for bin packing problem instance with anti-symmetry constraints.

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
$y_i = 0, i = \lceil \alpha \rceil, \ldots, m$	$y_i = 1, i = 1, \ldots, \lceil \alpha \rceil$

We can implement this branch in Dippy by writing a definition for the `branch_method`.

```
    if Item_antisymmetry:
        for n in range(0, len(bpp.ITEMS)):
    # Attach the problem data and variable dictionaries
    # to the DipProblem
```

```
    bpp         = prob.bpp
    assign_vars = prob.assign_vars
    tol         = prob.tol

    most   = float('-inf')
    assign = None
    for i in bpp.ITEMS:
            up   = ceil(sol[assign_vars[i, j]])  # Round up to next nearest integer
            down = floor(sol[assign_vars[i, j]]) # Round down
            frac = min(up - sol[assign_vars[i, j]], sol[assign_vars[i, j]] - down)
            if frac > tol: # Is fractional?
                    most = frac
                    assign = (i, j)

    down_lbs = {}
```

```
    down_ubs = {}
    up_lbs = {}
    up_ubs = {}
#    print assign, sol[assign_vars[assign]]
        down_ubs[assign_vars[assign]] = 0.0

        return down_lbs, down_ubs, up_lbs, up_ubs
```

The advanced branching decreases the size of the branch-and-bound tree further (see figure 4) and provides another symmetric solution with the bins used in order.

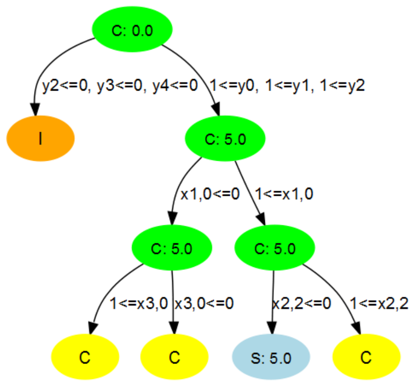

Figure 4: Branch-and-bound tree for bin packing problem instance with anti-symmetry constraints and advanced branching.

5.2 Adding Customised Cut Generation

By default DIP uses the CGL to add cuts. We can use `dippyOpts` to turn off CGL cuts and observe how effective the CGL are

```
def my_heuristics(prob, xhat, cost):
    sol = None
            sol = first_fit(prob)
    else:
        if prob.node_heuristic:
```

The branch-and-bound tree is significantly larger (see figure 5) than the original branch-and-bound tree that only used CGL cuts (see figure 2).

To add user-defined cuts in Dippy, we first define a new procedure for generating cuts and (if necessary) a procedure for determining a feasible solution. Within Dippy, this requires two new functions, `generate_cuts` and `is_solution_feasible`. As in §5.1, the embedded bin packing problem and decisions variables make it easy to access the solution values of variables in the bin packing problem. The inputs to `is_solution_feasible` are:

1. `prob` – the `DipProblem` being solved;

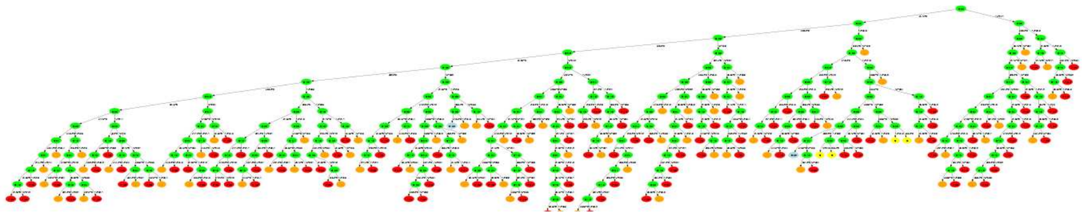

Figure 5: Branch-and-bound tree for bin packing problem instance without CGL cuts.

2. `sol` – an indexable object representing the solution at the current node;

3. `tol` – the zero tolerance value.

and the inputs to `generate_cuts` are:

1. `prob` – the `DipProblem` being solved;

2. `node` – various properties of the current node, including the solution.

If a solution is determined to be infeasible either by DIP (for example some integer variables are fractional) or by `is_solution_feasible` (which is useful for solving problems like the travelling salesman problem with cutting plane methods), cuts will be generated by `generate_cuts` and the in-built CGL (if enabled).

5.3 Adding Customised Column Generation

Using Dippy it is easy to transform a problem into a form that can be solved by either branch-and-cut or branch-price-and-cut. Branch-price-and-cut decomposes a problem into a master problem and a number of distinct subproblems. We can identify subproblems using the `relaxation` member of the `DipProblem` class. Once the subproblems have been identified, then they can either be ignored (when using branch-and-cut – the default method for DIP) or utilised (when using branch-price-and-cut – specified by turning on the `doPriceCut` option).

In branch-price-and-cut, the original problem is decomposed into a master problem and multiple subproblems [6]:

$$
\begin{array}{rllllll}
\min & c_1^\top x_1 & + c_2^\top x_2 & + & \cdots & + c_K^\top x_K & \\
\text{subject to} & A_1 x_1 & + A_2 x_2 & + & \cdots & + A_K x_K & = b \\
 & & F_2 x_2 & & & & = f_2 \\
 & & & & \ddots & & \vdots \\
 & & & & & F_K x_K & = f_K \\
\multicolumn{7}{c}{x_1 \in \mathbb{Z}^+_{n_1}, x_2 \in \mathbb{Z}^+_{n_2}, \ldots, x_K \in \mathbb{Z}^+_{n_K}}
\end{array}
\tag{1}
$$

In (1), there are $K-1$ subproblems defined by the constraints $F_k x_k = f_k, k \in 2, \ldots, K$. The constraints $A_1 x_1 + A_2 x_2 + \cdots + A_K x_K = b$ are known as *linking* constraints. Instead of solving (1) directly, column generation uses a convex combination of solutions y^k to each subproblem j to define the subproblem variables:

$$x_k = \sum_{l_k=1}^{L_k} \lambda_{l_k}^k y_{l_k}^k \tag{2}$$

where $0 \leq \lambda_{l_k}^k \leq 1$ and $\sum_{l_k=1}^{L_k} \lambda_{l_k}^k = 1$. By substituting (2) into the linking constraints and recognising that each $y_{l_k}^k$ satisfies $F_k x_k = f_k, x_k \in \mathbb{Z}_{n_k}^+$ (as it is a solution of this subproblem), we can form the *restricted* master problem (RMP) with corresponding duals ($\pi, \gamma_1, \ldots, \gamma_K$):

$$\begin{array}{llllllll}
\min & c_1^\top x_1 & +\sum_{l_2=1}^{L_2} \left(c_2^\top y_{l_2}^2\right) \lambda_{l_2}^2 & + \cdots & +\sum_{l_K=1}^{L_K} \left(c_K^\top y_{l_K}^K\right) \lambda_{l_K}^K & & & \\
\text{subject to} & A_1 x_1 & +\sum_{l_2=1}^{L_2} \left(A_2 y_{l_2}^2\right) \lambda_{l_2}^2 & + \cdots & +\sum_{l_K=1}^{L_K} \left(A_K y_{l_K}^K\right) \lambda_{l_K}^K & = b & : \pi & \\
& & \sum_{l_2=1}^{L_2} \lambda_{l_2}^2 & & & = 1 & : \gamma_1 & \\
& & & \ddots & & \vdots & & \\
& & & & \sum_{l_K=1}^{L_K} \lambda_{l_K}^K & = 1 & : \gamma_K & (3) \\
& & \sum_{l_2=1}^{L_2} y_{l_2}^2 \lambda_{l_2}^2 & & & \in \mathbb{Z}_{n_2}^+ & & \\
& & & \ddots & & \vdots & & \\
& & & & \sum_{l_K=1}^{L_K} y_{l_K}^K \lambda_{l_K}^K & \in \mathbb{Z}_{n_K}^+ & & \\
& x_1 \in \mathbb{Z}_{n_1}^+, \lambda^2 \in [0,1]_{L_2}, \ldots, \lambda^K & & & \in [0,1]_{L_K} & & &
\end{array}$$

The RMP provides the optimal solution $x_1^*, x_2^*, \ldots, x_K^*$ to the original problem (1) if the necessary subproblem solutions are present in the RMP. That is, if $y_{l_k}^{k,*}, l_k = 1, \ldots, L_k, k = 2, \ldots K$ such that $x_k^* = \sum_{l_k=1}^{L_k} \lambda_{l_k}^k y_{l_k}^{k,*}, k = 2, \ldots, K$ have been included.

Given that $x_k^*, k = 1, \ldots, K$ are not known a priori, column generation starts with an initial solution consisting of x_1 and initial sets of subproblem solutions. "Useful" subproblem solutions, that form columns for the RMP, are found by looking for subproblem solutions that provide columns with negative reduced cost. The reduced cost of a solution $y_{l_k}^k$'s column, i.e., the reduced cost for $\lambda_{l_k}^k$, is given by $c_k^\top y_{l_k}^k - \pi^\top A_k y_{l_k}^k - \gamma_k$. To find a solution with minimum reduced cost we can solve:

$$\begin{array}{lrl}
\mathcal{S}_k : \min & (c_k - \pi^\top A_k)^\top x_k - \gamma_k & \text{(reduced cost for corresponding } \lambda^k) \\
\text{subject to} & F_k x_k = f_k & \text{(ensures that } y^k \text{ solves subproblem } k) \\
& x_k \in \mathbb{Z}_{n_k}^+ &
\end{array} \tag{4}$$

If the objective value of $\mathcal{S}_k$ is less than 0, then the solution y^k will form a column in the RMP whose inclusion in the basis would improve the objective value of the RMP. The solution y^k is added to the set of solution used in the RMP. There are other mechanisms for managing the sets of solutions present in DIP, but they are beyond the scope of this paper.

Within DIP, hence Dippy, the RMP and *relaxed* problems $S_k, k = 2, \ldots, K$ are not specified explicitly. Rather, the constraints for each subproblem $F_k x_k = f_k$ are specified by using the `.relaxation[j]` syntax. DIP then automatically constructs the RMP and the relaxed problems $S_k, k = 2, \ldots, K$. The relaxed subproblems $S_k, k = 2, \ldots, K$ can either be solved using the default MILP solver (CBC) or a customised solver. A customised solver can be defined by the `relaxed_solver` function. This function has 4 inputs:

1. `prob` – the `DipProblem` being solved;
2. `index` – the index k of the subproblem being solved;
3. `redCosts` – the reduced costs for the x_k variables $c_k - \pi^\top A_k$;
4. `convexDual` – the dual value for the convexity constraint for this subproblem γ_k.

In addition to subproblem solutions generated using RMP dual values, initial columns for subproblems can also be generated either automatically using CBC or using a customised approach. A customised approach to initial variable generation can be defined by the `init_vars` function. This function has only 1 input, `prob`, the `DipProblem` being solved.

Starting from the original capacitated facility location problem from section 5:

$$\begin{array}{lll}
\min & \displaystyle\sum_{i=1}^{m} w_i & \\
\text{s.t.} & \displaystyle\sum_{i=1}^{m} x_{ij} = 1, j = 1, \ldots, n & \text{(each product produced)} \\
& \displaystyle\sum_{j=1}^{n} r_j x_{ij} + w_i = C y_i, i = 1, \ldots, m & \text{(aggregate capacity at location } i\text{)} \\
& x_{ij} \le y_i, i = 1, \ldots, m, j = 1, \ldots, n & \text{(disaggregate capacity at location } i\text{)} \\
& x_{ij} \in \{0,1\}, w_i \ge 0, y_i \in \{0,1\}, i = 1, \ldots, m, j = 1, \ldots, n &
\end{array}$$

we can decompose this formulation:

$$\begin{array}{lllllll}
\min & & & 1w_2 \cdots & & +1w_m & \\
\text{s.t.} & I\mathbf{x}_2 & & & \cdots +I\mathbf{x}_m & & = 1 \text{ (each product produced)} \\
& r^\top \mathbf{x}_2 & -Cy_2 & +1w_2 & & & = 0 \text{ (aggregate cap. at loc. 2)} \\
& I\mathbf{x}_2 & -ey_2 & & & & \le 0 \text{ (disaggregate cap. at loc. 2)} \\
& & & & \ddots & & \\
& & & & r^\top \mathbf{x}_m & -Cy_m + 1w_m & = 0 \text{ (aggregate cap. at loc. K)} \\
& & & & +I\mathbf{x}_m & -ey_m & \le 0 \text{ (disaggregate cap. at loc. K)}
\end{array}$$

where

$$\mathbf{x}_i = \begin{pmatrix} x_{i1} \\ \vdots \\ x_{in} \end{pmatrix}, r = \begin{pmatrix} r_1 \\ \vdots \\ r_n \end{pmatrix} \text{ and } e = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}.$$

Now the subproblems $F_k x_k = f_k, k = 2, \ldots, K$ are

$$\begin{bmatrix} r^\top & -C & 1 \\ I & e & \end{bmatrix} \begin{bmatrix} \mathbf{x}_i \\ y_i \\ w_i \end{bmatrix} \begin{matrix} = \\ \leq \end{matrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$c_k^\top = \left[\begin{array}{c|c|c} 0 & 0 & 1 \end{array}\right], A_k = \left[\begin{array}{c|c|c} I & 0 & 0 \end{array}\right],$$

so $\mathcal{S}_k$ becomes

$$\begin{array}{rllll} \mathcal{S}_i : \min & \sum_{j=1}^n -\pi_j x_{ij} & & +1w_i & -\gamma_i \\ \text{subject to} & \sum_{j=1}^n r_j x_{ij} & -Cy_i & +1w_i & = 0 \\ & x_{ij} & -y_i & & \leq 0, j = 1, \ldots, n \\ & x_{ij}, & y_i, \in \{0,1\}, j = 1, \ldots, n, w_i \geq 0 & & \end{array}$$

where π_j is the dual variable for the assignment constraint for product j in the RMP.

In Dippy, we define subproblems for each facility location using the `.relaxation` syntax for the aggregate and disaggregate capacity constraints:

```
# Aggregate capacity constraints
for i in LOCATIONS:
    prob.relaxation[i] += lpSum(assign_vars[(i, j)] * REQUIREMENT[j]
                                for j in PRODUCTS) + waste_vars[i] \
                                    == CAPACITY * use_vars[i]

# Disaggregate capacity constraints
for i in LOCATIONS:
    for j in PRODUCTS:
        prob.relaxation[i] += assign_vars[(i, j)] <= use_vars[i]
```

All remaining constraints (the assignment constraints that ensure each product is assigned to a facility) form the master problem when using branch-price-and-cut. To use branch-price-and-cut we turn on the `doPriceCut` option:

```
    'TolZero': '%s' % tol,
    'doPriceCut': '1',
    'generateInitVars': '1', })
```

Note that symmetry is also present in the decomposed problem, so we add ordering constraints (described in §5.1) to the RMP :

```
# Ordering constraints
for index, location in enumerate(LOCATIONS):
    if index > 0:
        prob += use_vars[LOCATIONS[index-1]] >= use_vars[location]
```

Using branch-price-and-cut, the RMP takes about ten times as long to solve as the original formulation, and has a search tree size of 37 nodes. The `generateInitVars` option uses CBC by default to find initial columns for the RMP and then uses CBC to solve the relaxed problems. Dippy lets us provide our own approaches to solving the relaxed problems and generating initial variables, which may be able to speed up the overall solution process.

In the relaxed problem for location i, the objective simplified to $\min \sum_{j=1}^{n} -\pi_j x_{ij} + 1w_i - \gamma_i$. However, the addition of the ordering constraints and the possibility of a Phase I/Phase II approach in the MILP solution process to find initial variables mean that our method must work for any reduced costs, i.e., the objective becomes $\min \sum_{j=1}^{n} d_j x_{ij} + f y_i + g w_i - \gamma_i$. Although the objective changes, the constraints remain the same. If we choose not to use a location, then $x_{ij} = y_i = w_i = 0$ for $j = 1, \ldots, n$ and the objective is $-\gamma_i$. Otherwise, we use the location and $y_i = 1$ and add f to the objective. The relaxed problem reduces to:

$$\begin{array}{rll} \min & \sum_{j=1}^{n} d_j x_{ij} & + g w_i - \gamma_i \\ \text{subject to} & \sum_{j=1}^{n} r_j x_{ij} & + 1 w_i = C \\ & x_{ij}, & w_i \in \{0,1\}, j = 1, \ldots, n \end{array}$$

However, the constraint ensures $w_i = C - \sum_{j=1}^{n} r_j x_{ij}$, so we can reformulate as:

$$\begin{array}{rl} \min & \sum_{j=1}^{n} (d_j - g r_j) x_{ij} \ + f C - \gamma_i \\ \text{subject to} & C - \sum_{j=1}^{n} r_j x_{ij} \geq 0 \Rightarrow \sum_{j=1}^{n} r_j x_{ij} \leq C \\ & x_{ij}, \in \{0,1\}, j = 1, \ldots, n \end{array}$$

This is a 0-1 knapsack problem with "effective costs" costs for each product j of $d_j - g r_j$. We can use dynamic programming to find the optimal solution.

In Dippy, we can access the problem data, variables and their reduced costs, so the 0-1 knapsack dynamic programming solution is straightforward to implement and use:

```python
def solve_subproblem(prob, index, redCosts, convexDual):
    loc = index

    # Calculate effective objective coefficient of products
    effs = {}
    for j in PRODUCTS:
        effs[j] = redCosts[assign_vars[(loc, j)]] \
                  - redCosts[waste_vars[loc]] * REQUIREMENT[j]

    avars = [assign_vars[(loc, j)] for j in PRODUCTS]
    obj = [-effs[j] for j in PRODUCTS]
    weights = [REQUIREMENT[j] for j in PRODUCTS]

    # Use 0-1 KP to max. total effective value of products at location
    z, solution = knapsack01(obj, weights, CAPACITY)
```

⋮

```
    rc = redCosts[use_vars[loc]] -z + \
         redCosts[waste_vars[loc]] * CAPACITY
    waste = CAPACITY - sum(weights[i] for i in solution)
    rc += redCosts[waste_vars[loc]] * waste

    # Return the solution if the reduced cost is low enough
    if rc < -tol: # The location is used and "useful"
        if rc - convexDual < -tol:
            var_values = [(avars[i], 1) for i in solution]
            var_values.append((use_vars[loc], 1))
            var_values.append((waste_vars[loc], waste))

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

    elif -convexDual < -tol: # An empty location is "useful"
            var_values = []

            dv = dippy.DecompVar(var_values, -convexDual, 0.0)
            return [dv]

    return []
```

Adding this customised solver reduces the solution time because it has the benefit of knowing it is solving a knapsack problem rather than a general MILP.

To generate initial facilities (complete with assigned products) we implemented two approaches. The first approach used a first-fit method and considered the products in order of decreasing requirement:

```
    # Sort the items in descending weight order
    productReqs = [(REQUIREMENT[j],j) for j in PRODUCTS]
    productReqs.sort(reverse=True)

    # Add items to locations, fitting in as much
    # as possible at each location.
    allLocations = []
    while len(productReqs) > 0:
        waste = CAPACITY
        currentLocation = []
        j = 0
        while j < len(productReqs):
            # Can we fit this product?
            if productReqs[j][0] <= waste:
                currentLocation.append(productReqs[j][1]) # index
                waste -= productReqs[j][0] # requirement
                productReqs.pop(j)
            else:
                # Try to fit next item
                j += 1
        allLocations.append((currentLocation, waste))
    # Return a list of tuples: ([products],waste)
    return allLocations
```

```
    locations = first_fit_heuristic()
    bvs = []
    index = 0
    for loc in locations:
        i = LOCATIONS[index]
        var_values = [(assign_vars[(i, j)], 1) for j in loc[0]]
        var_values.append((use_vars[i], 1))
        var_values.append((waste_vars[i], loc[1]))
        dv = dippy.DecompVar(var_values, None, loc[1])
        bvs.append((i, dv))
        index += 1
    return bvs
```

The second approach simply assigned one product to each facility:

```
    bvs = []
    for index, loc in enumerate(LOCATIONS):
        lc = [PRODUCTS[index]]
        waste = CAPACITY - REQUIREMENT[PRODUCTS[index]]
        var_values = [(assign_vars[(loc, j)], 1) for j in lc]
        var_values.append((use_vars[loc], 1))
        var_values.append((waste_vars[loc], waste))

        dv = dippy.DecompVar(var_values, None, waste)
        bvs.append((loc, dv))
    return bvs
```

Using Dippy we can define both approaches at once and then define which one to use by setting the `init_vars` method:

```
##prob.init_vars = one_each
```

These approaches define the initial sets of subproblem solutions $y^k_{l_k}, l_k = 1, \ldots, L_k, k = 1, \ldots, K$ for the initial RMP before the relaxed problems are solved using the RMP duals.

The effect of the different combinations of column generation, customised subproblem solvers and initial variable generation methods, both by themselves and combined with branching, heuristics, etc are summarised in Table 2. For this size of problem, column generation does not reduce the solution time significantly (if at all). However, we show in section 6 that using column branching enables DIP (via Dippy and PuLP) to be competitive with state-of-the-art solvers.

5.4 Adding Customised Heuristics

To add user-defined heuristics in Dippy, we first define a new procedure for node heuristics, `heuristics`. This function has three inputs:

1. `prob` – the `DipProblem` being solved;
2. `xhat` – an indexable object representing the fraction solution at the current node;
3. `cost` – the objective coefficients of the variables.

Multiple heuristics can be executed and all heuristic solutions can be returned to DIP.

```
#       print "# bins =", alpha
    up    = int(ceil(alpha))  # Round up to next nearest integer
    down  = int(floor(alpha)) # Round down
    frac  = min(up - alpha, alpha - down)
    if frac > tol: # Is fractional?
#     print "Symmetry branch"

        down_lbs = {}
        down_ubs = {}
        up_lbs = {}
        up_ubs = {}
        for n in range(up - 1, len(bpp.BINS)):
            down_ubs[use_vars[bpp.BINS[n]]] = 0.0
#           print down_ubs
```

A heuristic that solves the original problem may not be as useful when a fractional solution is available, so we demonstrate two different heuristics here: a "first-fit" heuristic and a "fractional-fit" heuristic.

In the facility location problem, an initial allocation of production to locations can be found using the same first-fit heuristic that provided initial solutions for the column generation approach (see §5.3). The first-fit heuristic iterates through the items requiring production and the facility locations allocating production at the first facility that has sufficient capacity to produce the item. This can then be used to provide an initial, feasible solution at the root node within the customised `heuristics` function.

```
#                  'LogDebugLevel': 5,
#                 'LogDumpModel': 5,

    status, message, primals, duals = dippy.Solve(prob, dippyOpts)

    if status == LpStatusOptimal:
        return dict((var, var.value()) for var in prob.variables())
    else:
        return None

def most_frac_use(prob, sol):
    # Get the attached data and variable dicts
    bpp         = prob.bpp
    use_vars    = prob.use_vars
    tol         = prob.tol

    most   = float('-inf')
    bin = None
    for j in bpp.BINS:
        alpha = sol[use_vars[j]]
        up    = ceil(alpha)  # Round up to next nearest integer
        down = floor(alpha) # Round down
        frac = min(up - alpha, alpha - down)
```

At each node in the branch-and-bound tree, the fractional solution (provided by `xhat`) gives an indication of the best allocation of production. One heuristic approach to “fixing” the fractional solution is to consider each allocation (of an item’s production to a facility) in order of decreasing fractionality and use a first-fit approach.

```
            if frac > most:
                most = frac
                bin = j

    down_lbs = {}
    down_ubs = {}
    up_lbs = {}
    up_ubs = {}
    if bin is not None:
#        print bin, sol[use_vars[bin]]
        down_ubs[use_vars[bin]] = 0.0
        up_lbs[use_vars[bin]] = 1.0

        return down_lbs, down_ubs, up_lbs, up_ubs

def most_frac_assign(prob, sol):
    # Get the attached data and variable dicts
    bpp         = prob.bpp
    assign_vars = prob.assign_vars
    tol         = prob.tol

    most   = float('-inf')
    assign = None
    for i in bpp.ITEMS:
        for j in bpp.BINS:
            up   = ceil(sol[assign_vars[i, j]])  # Round up to next nearest integer
            down = floor(sol[assign_vars[i, j]]) # Round down
            frac = min(up - sol[assign_vars[i, j]], sol[assign_vars[i, j]] - down)
            if frac > tol: # Is fractional?
                if frac > most:
                    most = frac
                    assign = (i, j)

    down_lbs = {}
    down_ubs = {}
    up_lbs = {}
    up_ubs = {}
    if assign is not None:
#    print assign, sol[assign_vars[assign]]
        down_ubs[assign_vars[assign]] = 0.0
        up_lbs[assign_vars[assign]] = 1.0

        return down_lbs, down_ubs, up_lbs, up_ubs

def symmetry(prob, sol):
    # Get the attached data and variable dicts
    bpp      = prob.bpp
    use_vars = prob.use_vars
    tol      = prob.tol
```

Running the first-fit heuristic before starting the branching process has little effect on the solution time and does not reduce the number of nodes. Adding the first-fit heuristic guided by fractional values increases the solution time slightly and the number of nodes remains at 419. The reason this heuristic was not that helpful for this problem instance is that:

- the optimal solution is found within the first 10 nodes without any heuristics, so the heuristic only provides an improved upper bound for < 10 nodes;
- the extra overhead of the heuristic at each node increases the solution time more than any decrease from exploring fewer nodes.

5.5 Combining Techniques

The techniques and modifications of the solver framework can be combined to improve performance further. Table 2 shows that it is possible to quickly and easily test many approaches for a particular problem, including combinations of approaches[2]. Looking at the results shows that the heuristics only help when the size of the branch-and-bound tree has been reduced with other approaches, such as ordering constraints and advanced branching. Approaches for solving this problem that warrant further investigation use column generation, the customised solver and either ordering constraints or the first-fit heuristic to generate initial variables. Tests with different data showed that the solution time for branch-price-and-cut doesn't increase with problem size as quickly as for branch-and-cut, so the column generation approaches are worth considering for larger problems.

6 Performance and Conclusions

In section 5 we showed how Dippy works in practice by making customisations to the solver framework for an example problem. We will use the Wedding Planner problem from the PuLP documentation [3] to compare Dippy to two leading solvers that utilise branch-and-cut: the open-source CBC and the commercial Gurobi. This particular problem is useful for comparing performance because it has a natural column generation formulation and can be scaled-up in a simple way, unlike the Facility Location problem which is strongly dependent on the specific instance being tested.

The Wedding Planner problem is as follows: given a list of wedding attendees, a wedding planner must come up with a seating plan to minimise the unhappiness of all of the guests. The unhappiness of guest is defined as their maximum unhappiness at being seated with each of the other guests at their table, making it a pairwise function. The unhappiness of a table is the maximum unhappiness of all the guests at the table. All guests must be seated and there is a limited number of seats at each table.

This is a set partitioning problem, as the set of guests G must be partitioned into multiple subsets, with the members of each subset seated at the same table. The cardinality of the subsets is determined by the number of seats at a table and

[2]All tests were run using Python 2.7.1 on a Windows 7 machine with an Intel Core 2 Duo T9500@2.60GHz CPU.

the unhappiness of a table can be determined by the subset. The MILP formulation is:

$$
\begin{aligned}
x_{gt} &= \begin{cases} 1 & \text{if guest } g \text{ sits at table } t \\ 0 & \text{otherwise} \end{cases} \\
u_t &= \text{unhappiness of table } t \\
S &= \text{number of seats at a table} \\
U(g,h) &= \text{unhappiness of guests } g \text{ and } h \text{ if they are seated at the same table}
\end{aligned}
$$

$$
\begin{aligned}
\min \quad \sum_{t \in T} u_t & \quad \text{(total unhappiness of the tables)} \\
\sum_{g \in G} x_{gt} & \leq S, t \in T \\
\sum_{t \in T} x_{gt} & = 1, g \in G \\
u_t & \geq U(g,h)(x_{gt} + x_{ht} - 1), t \in T, g < h \in G
\end{aligned}
$$

Since DIP, and thus Dippy, doesn't require a problem to be explicitly formulated as a Dantzig-Wolfe decomposition, a change from DIP to CBC is trivial. The only differences are that:

1. A `LpProblem` is created instead of a `DipProblem`;

2. No `.relaxation` statements are used;

3. The `LpProblem.solve` method uses CBC to solve the problem.

To see if CBC and Gurobi would perform well with a column-based approach, we also formulated a problem equivalent to the restricted master problem from the branch-price-and-cut approach and generated and added all possible columns before the solving the MILP. Finally we used to Dippy to develop a customised solver and initial variable generation function for the branch-price-and-cut formulation in DIP. In total, six approaches were tested on problem instances of increasing size:

1. CBC called from PuLP;

2. CBC called from PuLP using a columnwise formulation and generating all columns a priori;

3. Gurobi called from PuLP;

4. Gurobi called from PuLP using a columnwise formulation and generating all columns a priori;

5. DIP called from Dippy using branch-price-and-cut without customisation;

6. DIP called from Dippy using customised branching, cuts and column generation callback functions.

In Table 3 and Figure 6 we see that[3]:

- Gurobi is fastest for small problems;
- The symmetry present in the problem means the solution time of CBC and Gurobi for the original problem deteriorate quickly;
- The time taken to solve the columnwise formulation also deteriorates, but at a lesser rate than when using CBC or Gurobi on the original problem;
- Both DIP and customised DIP solution times grow at a lesser rate than any of the CBC/Gurobi approaches;
- For large problems, DIP becomes the preferred approach.

The main motivation for the development of Dippy was to alleviate obstacles to experimentation with and customisation of advanced MILP frameworks. These obstacles arose from an inability to use the description of a problem in a high-level modelling languag integrated with the callback functions in leading solvers. This is mitigated with Dippy by using the Python-based modelling language PuLP to describe the problem and then exploiting Python's variable scoping rules to implement the callback functions.

Using the Capacitated Facility Location problem we have shown that Dippy is relatively simple to experiment with and customise, enabling the user to quickly and easily test many approaches for a particular problem, including combinations of approaches. In practice Dippy has been used successfully to enable final year undergraduate students to experiment with advanced branching, cut generation, column generation and root/node heuristics. The Wedding Planner problem shows that Dippy can be a highly competitive solver for problems in which column generation is the preferred approach. Given the demonstrated ease of the implementation of advanced MILP techniques and the flexibility of a high-level mathematical modelling language, this suggests that Dippy is effective as more than just an experimental "toy" or educational tool. It enables users to concentrate on furthering Operations Research knowledge and solving hard problems instead of spending time worrying about implementation details. Dippy breaks down the barriers to experimentation with advanced MILP approaches for both practitioners and researchers.

[3] All tests were run using Python 2.7.1 on a Dell XPS1530 laptop with an Intel Core 2 Duo CPU T9500@2.60GHz and 4 GB of RAM. We used CBC version 2.30.00, Gurobi version 4.5.1, and Dippy version 1.0.10.

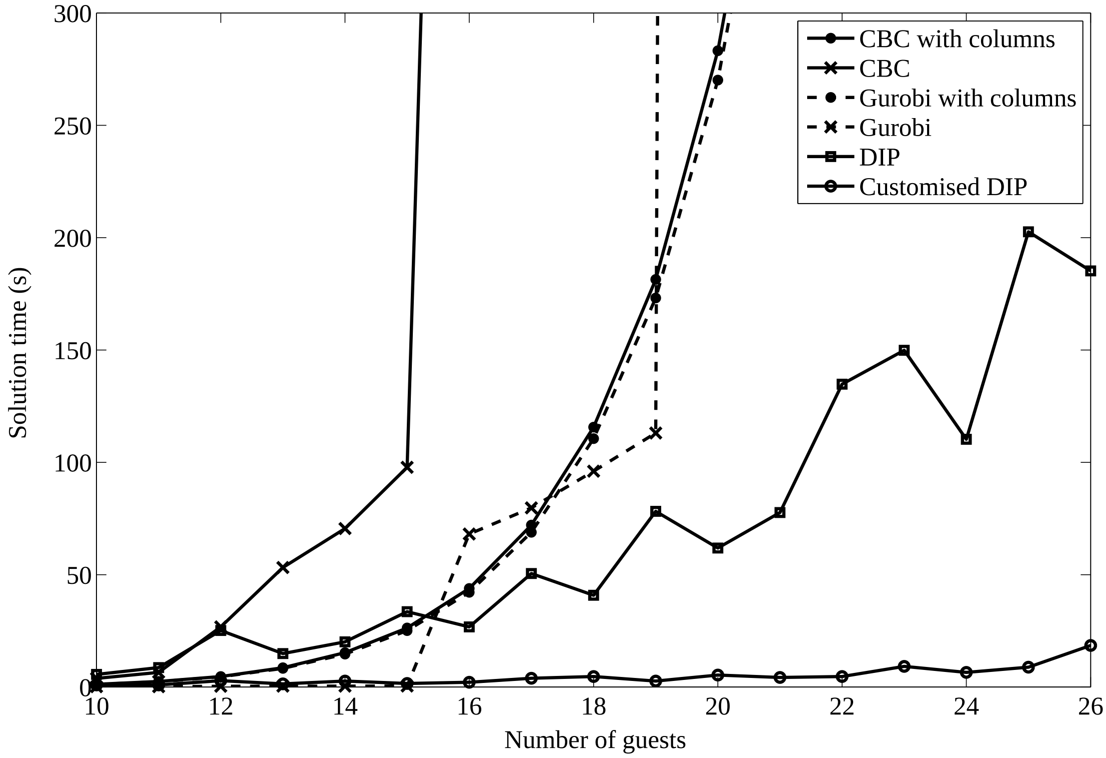

Figure 6: Comparing solver performance on the Wedding Planner problem. In this figure the times for generating the columns for "CBC with columns" and "Gurobi with columns" have been included in the total solve time. The time required for solving the original formulation sharply increases for both Gurobi and CBC (marked with crosses) but at different problem sizes. However the time for the column-wise formulation is similar for Gurobi and CBC. The time for DIP does not smoothly increase with problem size, but is consistently lower than Gurobi for instances with 16 or more guests.

7 Acknowledgments

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

References

[1] G. Leyland, S. Kraines, T. Akatsuka, A. K. Molyneaux, D. Favrat, and H. Komiyama. Minimising transport and investment costs in the coke industry in shanxi province, china, 2002. In preparation.

[2] R. Lougee-Heimer. The Common Optimization Interface for Operations Research. *IBM Journal of Research and Development*, 47(1):57–66, 2003.

[3] S. A. Mitchell and J.S. Roy. PuLP.

[4] Michael J. O'Sullivan and Cameron G. Walker. The Sponge Roll Production Problem, 2008. Case study within the online teaching TWiki (Department of Engineering Science, University of Auckland), `http://twiki.esc.auckland.ac.nz/twiki/bin/view/OpsRes/SpongeRollProductionProblem`.

[5] T. K. Ralphs and M. V. Galati. Decomposition in integer programming. In J Karlof, editor, *Integer Programming: Theory and Practice*. CRC Press, 2005.

[6] François Vanderbeck. On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Oper. Res.*, 48:111–128, 2000.

[7] Cameron G. Walker and Michael J. O'Sullivan. The Coke Supply Chain Problem, 2008. Case study within the online teaching TWiki (Department of Engineering Science, University of Auckland), `http://twiki.esc.auckland.ac.nz/twiki/bin/view/OpsRes/CokeSupplyChain`.

Strategies	Time (s)	Nodes
Default (branch and cut)	0.26	419
+ ordering constraints (OC)	0.05	77
+ OC & advanced branching (AB)	0.01	3
+ weighted inequalities (WI)	0.34	77
+ WI & OC	0.17	20
+ WI & OC & AB	0.06	4
+ first-fit heuristic (FF) at root node	0.28	419
+ FF & OC	0.05	77
+ FF & OC & AB	0.01	3
+ FF & WI	0.36	77
+ FF & WI & OC	0.14	17
+ FF & WI & OC & AB	0.05	3
+ fractional-fit heuristic (RF) at nodes	0.28	419
+ RF & OC	0.05	77
+ RF & OC & AB	0.01	3
+ WI & RF	0.38	77
+ WI & RF & OC	0.14	17
+ WI & RF & OC & AB	0.05	3
+ FF & RF	0.28	419
+ FF & RF & OC	0.05	77
+ FF & RF & OC & AB	0.01	3
+ WI & FF & RF	0.38	77
+ WI & FF & RF & OC	0.14	17
+ WI & FF & RF & OC & AB	0.05	3
+ column generation (CG)	2.98	37
+ CG & OC	2.07	23
+ CG & OC & AB	0.56	10
+ CG & customised subproblem solver (CS)	2.87	37
+ CG & CS & OC	1.95	23
+ CG & CS & OC & AB	0.44	10
+ CG & first-fit initial variable generation (FV)	3.96	45
+ CG & CS & FV	3.72	45
+ CG & CS & FV & OC	1.70	18
+ CG & CS & FV & OC & AB	0.22	3
+ CG & one-each initial variable generation (OV)	3.40	41
+ CG & CS & OV	3.33	41
+ CG & CS & OV & OC	2.23	24
+ CG & CS & OV & OC & AB	0.27	3

Table 2: Experiments for the Capacitated Facility Location Problem

# guests	Time (s)								
	CBC	CBC & columns		Gurobi	Gurobi & columns		DIP	Customised
		gen vars	solve		gen vars	solve		DIP
6	0.07	0.01	0.06	0.04	0.01	0.05	0.90	0.33
7	0.07	0.01	0.12	0.04	0.01	0.11	1.77	0.57
8	0.90	0.01	0.27	0.07	0.01	0.25	4.78	0.57
9	2.54	0.01	0.57	0.09	0.01	0.55	2.11	0.78
10	3.83	0.01	1.23	0.13	0.01	1.15	5.60	0.94
11	6.48	0.01	2.46	0.14	0.01	2.36	8.62	0.91
12	26.73	0.01	4.64	0.34	0.01	4.55	25.17	2.80
13	53.18	0.01	8.57	0.39	0.01	8.28	14.86	1.40
14	70.51	0.01	15.27	0.38	0.01	14.65	20.09	2.66
15	97.79	0.01	26.26	0.47	0.01	25.07	33.52	1.59
16	>1000	0.01	43.86	68.08	0.01	42.11	26.73	2.09
17	–	0.01	72.07	79.71	0.01	68.87	50.48	3.92
18	–	0.01	115.64	96.03	0.01	110.52	40.80	4.67
19	–	0.01	181.39	113.01	0.01	173.13	78.20	2.64
20	–	0.02	283.16	>6000	0.01	270.08	61.86	5.31
21	–	0.02	434.60	–	0.02	418.04	77.66	4.23
22	–	0.02	664.87	–	0.02	639.04	134.76	4.63
23	–	–	>1000	–	–	>1000	149.82	9.16
24	–	–	–	–	–	–	110.24	6.51
25	–	–	–	–	–	–	202.59	8.80
26	–	–	–	–	–	–	185.21	18.47

Table 3: Experiments for the Wedding Planner Problem