Dantzig-Wolfe decomposition and branch-and-price solving in G12

Jakob Puchinger, Peter Stuckey, Mark Wallace, Sebastian Brand

Constraints manuscript No.
(will be inserted by the editor)

Dantzig-Wolfe Decomposition and Branch-and-Price Solving in G12

Jakob Puchinger[1], **Peter J. Stuckey**[2],
Mark Wallace[3], **Sebastian Brand**[2]

[1] Austrian Institute of Technology
Vienna, Austria
e-mail: jakob.puchinger@ait.ac.at
[2] NICTA Victoria Research Laboratory
Department of Computer Science & Software Engineering
University of Melbourne, Australia
e-mail: {pjs,sbrand}@csse.unimelb.edu.au
[3] School of Computer Science and Software Engineering
Monash University, Melbourne, Australia
e-mail: mgw@mail.csse.monash.edu.au

The date of receipt and acceptance will be inserted by the editor

Abstract The G12 project is developing a software environment for stating and solving combinatorial problems by mapping a high-level model of the problem to an efficient combination of solving methods. Model annotations are used to control this process. In this paper we explain the mapping to branch-and-price solving. Dantzig-Wolfe decomposition is automatically performed using the additional information given by the model annotations. The models obtained can then be solved using column generation and branch-and-price. G12 supports the selection of specialised subproblem solvers, the aggregation of identical subproblems to reduce symmetries, automatic disaggregation when required by branch-and-bound, the use of specialised subproblem constraint-branching rules, and different master problem solvers including a hybrid solver based on the volume algorithm. We demonstrate the benefits of the G12 framework on three examples: a trucking problem, cutting stock, and two-dimensional bin packing.

1 Introduction

1.1 Mapping Models to Hybrid Algorithms

Combinatorial optimisation problems are easy to state, but hard to solve. They arise in a huge variety of applications. Branch-and-price is one of many

powerful methods for solving them. This paper describes how Dantzig-Wolfe decomposition, column generation and branch-and-price are integrated into the hybrid optimisation platform G12 [29]. The G12 project consists in developing a software environment for stating and solving combinatorial problems by mapping a high-level model of the problem to an efficient combination of solving methods. We call such a combination of methods a *hybrid algorithm*. Because there is no method for choosing the best way to solve a given problem, we believe the (human) problem solver must be able to experiment easily with different hybrid algorithms. To meet this purpose, the G12 project is developing user-controlled mappings from a high level model to different solving methods. These mappings must satisfy three conflicting objectives. They must be

- efficient, enabling the human problem solver to tightly control the behaviour of the algorithm if necessary for performance;
- flexible, allowing plug-and-play between different sub-algorithms;
- easy-to-use and easy-to-change for efficient experimentation with alternative hybrid algorithms.

The mapping to branch-and-price presented in this paper is designed to meet all three objectives (in reverse order):

- The user can select branch-and-price and control its behaviour by annotating a high-level model of the problem.
- The generated algorithm can use a separate solver for the subproblem. The user can control the decomposition and select the subproblem solver by further annotations.
- Inefficiencies arising as a result of identical subproblems are avoided by aggregating them, but the user is still enabled to express search control in terms of variables in the original model. The system also supports specialised branching rules, allowing fine-grained control of search where necessary.

1.2 The G12 Platform

The G12 platform consists of three major components, the high-level modelling language Zinc [13], the model transformation language Cadmium [10], and several internal and external solvers written and/or interfaced using the general-purpose programming language Mercury [28].

The system allows one to take a model written in Zinc, transform it to various underlying solvers using Cadmium, and then execute it. Cadmium transformations can be selected from a library of standard transformations or can be user-defined. Mappings from Zinc to Finite Domain Constraint Programming (FD) or Linear Programming (LP) models are available [6]. To control these transformations the user can annotate the model.

At the solver programming language (Mercury) level, G12 defines interfaces to solvers such as an FD solver, a continuous interval constraint

solver, and LP solvers using type classes. Various implementations of these interfaces are provided, e.g. for LP/MIP solvers including CPLEX and COIN-OR/OSI. The Dantzig-Wolfe decomposition column generation, default branch-and-bound, and branch-and-price solvers heavily rely on the LP solver interfaces. These interfaces provide standard predicates for variable creation, constraint posting, setting an objective function, and LP and MIP optimisation. This system of pluggable components allows us to quickly design new hybrid algorithms and to combine existing solvers in innovative ways.

1.3 Plan of the Paper

We discuss related work before introducing Dantzig-Wolfe decomposition and column generation. We then explain how these techniques are used in the G12 system and present computational experiments on a trucking problem. Thereafter we describe several more advanced features of our system. We explain how the G12 column generation implementation deals with identical subproblems. We use the cutting stock problem as an example and present results of some computational experiments. We then elaborate how a hybrid volume algorithm/linear programming solver can be used to solve the column generation master problem; further computational experiments are performed on the cutting stock problem. We then show how our system allows for the implementation of specialised branching rules and report computational experiments for the two-dimensional bin packing problem. Finally conclusions are drawn and some future research directions are pointed out.

2 Related Work

The practical usefulness of column generation and branch-and-price has been well-established over the last 20 years [9,4]. More recently it has emerged that column generation provides an ideal method for combining approaches, such as constraint programming, local search, and integer/linear programming. Columns can be generated by constraint programming or application-specific algorithms, while the master problem is handled using branch-and-price [18,35,26,24].

For systems such as G12 that support hybrid algorithms, Dantzig-Wolfe decomposition, column generation and branch-and-price provide an elegant way for the different solving techniques to be combined. However, the specification of this form of hybrid is quite complex, as it requires adaptation of simplex-based approaches to support the lazy generation of columns. Thus systems such as ABACUS [17], MINTO [22], OPL script [30], MAESTRO [7], COIN/BCP [25], SCIP [1], and COMET [31] offer facilities to support the implementation of column generation or branch-and-price on top of generic integer/linear programming packages. However, these systems

still require the user to understand the technical details of branch-and-price: the purpose was to support algorithm implementation rather than problem modelling. Both COMET and OPL allow column generation to be specified in an elegant way. Example code for the cutting stock problem is available in [30] for OPL and in the COMET distribution[1]. However, these examples are not comparable to the approaches developed in our work, since no branching in the master problem is performed. The examples are therefore not implementing branch-and-price and cannot ensure that optimal solutions are obtained as final results. Furthermore, they cannot guarantee that feasible integer solutions are obtained if the master problem is not a set covering problem where feasible solutions can simply be obtained by rounding.

A system based on similar ideas such as annotating high-level models is SIMPL [34], allowing various types of hybrid solving. The focus of SIMPL is on instances of a branch-infer-relax model of solving which naturally fits with Benders decomposition and branch-and-bound search. The paper [34] allows a `bp` search specification indicating branch-and-price search, but there are no details of how this is supported, nor any examples making use of it.

Certainly column generation is technical, but for people trying to solve combinatorial problems the most important requirement is to be able to try out an algorithm, or more generally a hybrid algorithm, quickly and easily without rewriting the problem specification. The first attempt to provide a column generation library was in ECL^{i}PS^e [11,12]. It introduced the idea of an aggregate variable appearing in the master problem to represent a set of values returned as columns from multiple solutions to identical subproblems. However this library assumes a fixed set of variables in each subproblem, and precludes search choices which break some of the subproblem symmetries. In order to achieve tight control over branch-and-price, sophisticated ECL^{i}PS^e users have required special adaptations of the column generation library in order to be able to work directly with low level primitives [23].

3 Dantzig-Wolfe Decomposition, Column Generation and Branch-and-Price

Dantzig-Wolfe decomposition is a standard way to decompose an integer programming model into a master problem and one or several subproblems [8,21,9]. The bound on the objective resulting from the LP relaxation of the decomposed model is usually stronger than that of the original formulation (if the subproblem does not have the integrality property). This can result in a smaller search space in LP-based branch-and-bound algorithms.

[1] `http://www.comet-online.org`

The *Original Problem* has the form

$$\begin{array}{lll} \mathsf{OP}: & \text{minimise} & \sum_{k\in K} c^k x^k \\ & \text{subject to} & \sum_{k\in K} A^k_j x^k \geq b_j \qquad \forall j = 1\ldots M \\ & & x^k \in \mathcal{D}^k \qquad\qquad\qquad k \in K. \end{array}$$

Since we are considering pure integer subproblems, the $\mathcal{D}^k$ are finite sets of vectors in $\mathbb{Z}_+^{N_k}$ implicitly defined by additional constraints. We view the elements of $\mathcal{D}^k$ as indexed using an index set P^k; that is, we have $\mathcal{D}^k = \{d^k_p \mid p \in P^k\}$. We can then alternatively write

$$\mathcal{D}^k = \{e^k \in \mathbb{R}^{N_k} \mid e^k = \sum\nolimits_{p\in P^k} d^k_p \lambda^k_p,\ \sum\nolimits_{p\in P^k} \lambda^k_p = 1;\ \lambda^k_p \in \{0,1\}\ \forall p \in P^k\}.$$

Substituting the x^k by the λ^k_p in OP, we obtain the *Master Problem*:

$$\begin{array}{lll} \mathsf{MP}: & \text{minimise} & \sum_{k\in K}\sum_{p\in P^k} c^k d^k_p \lambda^k_p \\ & \text{subject to} & \sum_{k\in K}\sum_{p\in P^k} A^k_j d^k_p \lambda^k_p \geq b_j \qquad \forall j = 1\ldots M \quad (1) \\ & & \sum_{p\in P^k} \lambda^k_p = 1 \qquad\qquad\qquad k \in K \quad (2) \\ & & \lambda^k_p \in \{0,1\} \qquad\qquad \forall p \in P^k,\ k \in K. \end{array}$$

Dantzig-Wolfe decomposition typically results in a master problem with many variables. To deal with a possibly exponential number of variables, delayed column generation [9] is used. Starting from a restricted LP-relaxation of the original problem, the *Restricted Master Problem*, variables (columns) are dynamically included in order to find an optimal solution.

The simplex algorithm for solving linear programs proceeds from one basic feasible solution to the next one, always in direction of a potential improvement of the objective function. This is achieved by adding a variable with negative reduced cost to the basis and by removing some other variable from it. Reduced costs can be seen as an optimistic estimate of the amount of improvement achieved by a unit increase of their corresponding variable. This is the crucial property of the simplex algorithm exploited in column generation. For every $\mathcal{D}^k$, a subproblem is solved to identify a subproblem solution d^k_p with negative reduced cost

$$(c^k - \pi A^k) d^k_p - \mu^k \tag{3}$$

where π are the dual variable values corresponding to the constraints (1) and μ^k is the dual value of the kth convexity constraint (2). If a column with negative reduced cost exists, it is not necessary to look for a column with minimal negative reduced cost.

Since the column generation algorithm alone only solves the LP-relaxed version of the problem, one has to branch to guarantee integrality of the

variables. A standard linear programming based branch-and-bound algorithm branches on the original variables. This does not affect the subproblem structure [33]. Adding a constraint to the original problem corresponds to adding a row to MP:

$$\sum_{k \in K} \sum_{p \in P^k} C^k d_p^k \lambda_p^k \geq c.$$

This additional row affects only the coefficients of the objective function of the subproblem (3), which becomes

$$(c^k - \pi A^k - \nu C^k) d^k - \mu^k,$$

where ν is the dual variable value corresponding to the newly added constraint.

The additional branching constraint could render the restricted master problem infeasible. But, since one usually does not deal with the complete master problem, additional columns can possibly restore feasibility of the restricted master problem. Such columns are obtained by solving a problem very similar to the pricing problem [16]. The infeasibility of an LP is proven with the existence of a dual ray, describing an unbounded direction of the dual problem. In order to restore feasibility, a column whose inner product with the dual ray is positive, has to be determined. This column yields a new constraint in the dual problem removing its unboundedness along the direction of the given dual ray. Solving a problem very similar to the standard subproblem, where the dual ray is used instead of the optimal dual variable values and disregarding the original objective function coefficients, yields such a column or a proof that no such column exists. Only in the latter case can one conclude that the branching constraint rendered the MP infeasible.

4 Zinc Formulations of Dantzig-Wolfe Decomposition

4.1 Solver and Search Annotations

To use Dantzig-Wolfe decomposition and column generation in G12, one annotates a high-level Zinc model to explain what parts define the master problem and the subproblems and which solvers are to be used for them. We demonstrate this by a simple transportation problem.

The Trucking problem. Consider the following problem, inspired by [5]. We are given `N` trucks; each truck has a cost and an amount of material it can load. We are further given `P` time periods; in each time period a given demand of material has to be shipped. Each truck also has constraints on usage: in each consecutive k time periods it must be used at least l and at most u times. The Zinc model of the problem is as follows:

Trucking.zinc - user defined

```
int: P;                                          type Periods = 1..P;
int: T;                                          type Trucks = 1..T;
array[Periods] of int: Demand;                   array[Trucks] of int: Cost;
array[Trucks] of int: Load;                      array[Trucks] of int: K;
array[Trucks] of int: L;                         array[Trucks] of int: U;
array[Periods] of var set of Trucks: x;

constraint
    forall(p in Periods)( sum_set(x[p], Load) >= Demand[p] );

constraint
    forall(t in Trucks)(
        sequence([ bool2int(t in x[p]) | p in Periods ], L[t], U[t], K[t])
    );

solve minimize sum(p in Periods)( sum_set(x[p], Cost) );
```

At each time period we need to choose which trucks to use in order to ship enough material and satisfy the usage limits. The `sum_set`(s, f) function returns $\Sigma_{e \in s} f(e)$, while the `sequence`$([y_1, \ldots, y_n], l, u, k)$ constrains the sum of each subsequence of length k, $y_i + \cdots + y_{i-k-1}, 1 \leq i \leq n - k + 1$ to be between l and u inclusive. The `bool2int` function coerces a Boolean to `0..1`. As it stands this model is directly executable in an FD solver that supports set variables. There exist specialised propagators for `sum_set` and `sequence`.

In Zinc we can control the search by adding an *annotation* on the solve item. For example,

```
solve :: set_search(x, first_fail, outdomain_min, complete)
    minimize sum(p in Periods)( sum_set(x[p], Cost) );
```

indicates that we want label the set variables with smallest domain first (*first_fail*) by first trying to exclude the least unknown element of the set and then including it (*outdomain_min*) in a complete search.

A specific set of annotations is used to specify branch-and-price models:

colgen_subproblem_constraint(int: id, ann: solver) designates an individual subproblem and associates it with an identifier and the solver to be used for it,

colgen_solver(ann: solver) specifies the solver to be used for the master problem,

lp_bb(array[int] of var int: x, ann: choice, ann: split) specifies the search strategy as LP-based branch-and-bound on the given variables with specific variable choice and split methods.

The trucking problem example shows these annotations in use (unchanged model components are omitted):

Trucking.zinc - user defined

```
...

constraint
    forall(p in Periods)(
        (sum_set(x[p], Load) >= Demand[p])
```

```
            :: colgen_subproblem_constraint(p, mip)
    );

solve :: colgen_solver(lp)
      :: lp_bb(x, most_frac, std_split)
    minimize sum(p in Periods)( sum_set(x[p], Cost) );
```

For each `Period` a subproblem is defined in terms of its constraints and solver. Note that we could have used a more specialised solver here since the subproblem is a knapsack problem. The annotation to the `solve` item determines the solver for the master problem as well as the search specification, branch-and-bound selecting the most fractional variable first and performing a standard split ($x \leq \lfloor x^{LP} \rfloor$ or $x \geq \lceil x^{LP} \rceil$).

4.2 Automatic Model Reformulation

The annotated model is subjected to an automatic, implicit Dantzig-Wolfe decomposition controlled by a Cadmium transformation. The resulting Zinc model is such that it can directly be handled by the G12 column generation module. The main components of this transformation are explained in the following.

Step 1: Variable declarations. First, the variables to be used in column generation are determined. They are those variables occurring in the subproblem constraint expressions that are indexed by the respective subproblem identifier. For each such original problem variable, a subproblem variable and a (so-called) master problem variable is declared. The master problem variables are place-holders representing the implicit sums of the λ variables $\sum_{p \in P^k} d_p^k \lambda_p^k$ as introduced in MP.

A second set of column generation specific low-level annotations are employed to differentiate the variables. They are automatically attached to variable declarations as appropriate:

colgen_var designates original variables,

colgen_master_var designates master variables,

colgen_subproblem_var(ann: solver) designates subproblem variables and the subproblem solver to be used.

While creating the additional variable declarations, the transformation keeps track of which original, master and subproblem variables are associated with each other. This information is conveyed to the G12 column generation module in the form of special `colgen_link` constraints, which are automatically added to the Zinc model. They allow the original variable values to be recovered from the master problem, and they link the subproblem variables to the original variables so that the correct subproblem objectives can be derived by the column generation module.

In the trucking example, the subproblem variables are the `x[p]` where `p` is the subproblem identifier. Corresponding additional variables `mx[p]` and `sx[p]` are automatically introduced.

Step 2: Model partitioning. Next, the constraints are turned into either master problem or subproblem constraints by replacing the original variables appropriately. In subproblem constraints, they are replaced by the corresponding subproblem variables, while generally elsewhere they are replaced by the corresponding master problem variables. However note that the search is still expressed in terms of the original problem variables.

Here is the resulting generated model of the trucking problem:

```
———— Trucking_2.zinc - automatically generated by steps 1,2 ————
...

array[Periods] of var set of Trucks:  x :: colgen_var;
array[Periods] of var set of Trucks: mx :: colgen_master_var;
array[Periods] of var set of Trucks: sx :: colgen_subproblem_var(mip);

constraint
    forall(p in Periods)(
        colgen_link(x[p], mx[p], sx[p])
    );

constraint
    forall(p in Periods)(
        (sum_set(sx[p], Load) >= Demand[p])
            :: colgen_subproblem_constraint(p, mip)
    );

constraint
    forall(t in Trucks)(
        sequence([ bool2int(t in mx[p]) | p in Periods ], L[t], U[t], K[t])
    );

solve :: colgen_solver(lp)
      :: lp_bb(x, most_frac, std_split)
    minimize sum(p in Periods)( sum_set(mx[p], Cost) );
————————————————————————————————————————————————————
```

Step 3: Linearisation and further solver-specific preprocessing. Since column generation is to be used, the master constraints and objective function must be linear. Subproblem solvers in general may accept the subproblem variables and constraints in their original representation, or they may require them in a solver-specific form. For example, if a MIP solver is specified, linearisation also of the subproblems is needed.

The master and subproblem constraints in our example can be linearised by giving suitable definitions for the `sum_set` and `sequence` globals, e.g.:

```
function var int: sum_set(var set of $T: s, array[$T] of int: cost) =
    sum(e in index_set(cost))( cost[e] * bool2int(e in s) );

predicate sequence(array[int] of var int: y, int: l, int: u, int: k) =
    forall(i in min(index_set(y)) .. max(index_set(y)) - k + 1)(
        let { var int: s = sum(j in i .. i + k - 1)( y[j] ) }
        in
        s >= l /\ s <= u
    );
```

The `index_set` function returns the set of indices of its array argument. Finally, a generic Cadmium transformation [6] can be used to transform the array of set variables `x` into a two-dimensional array of `0..1` variables such that `x[p,t] = 1` if `t` $\in$ `x[p]`.

The following is the final result of the reformulation:

```
———— Trucking_3.zinc - automatically generated by steps 1,2,3 ————
...

array[Trucks, Periods] of var 0..1:  x :: colgen_var;
array[Trucks, Periods] of var 0..1: mx :: colgen_master_var;
array[Trucks, Periods] of var 0..1: sx :: colgen_subproblem_var(mip);

constraint
    forall(t in Trucks, p in Periods)(
        colgen_link(x[t,p], mx[t,p], sx[t,p])
    );

constraint
    forall(p in Periods)(
        (sum(t in Trucks)( Load[t] * sx[t,p] ) >= Demand[p])
            :: colgen_subproblem_constraint(p, mip)
    );

constraint
    forall(t in Trucks)(
        forall(i in 1 ..  P - K[t] + 1)(
            let { var int: s = sum(j in i .. i + K[t] - 1)( mx[t,j] ) }
            in
            s >= L[t] /\ s <= U[t]
        )
    );

solve :: colgen_solver(lp)
      :: lp_bb(x, most_frac, std_split)
    minimize sum(p in Periods)( sum(t in Trucks)( Cost[t] * mx[t,p] ) );
```

5 Column Generation and Branch-and-Price in G12

A Zinc model obtained from the automatic reformulation is suitable to be sent to the column generation module.

5.1 Column Generation Module

The G12 column generation module reads the transformed model together with its instance data. It builds the subproblems and attaches them to the requested solvers. These solvers must support optimisation with a linear objective function, and preferably support it in an incremental way. Then the restricted master problem is defined and attached to a solver that supports delayed column generation: currently LP solvers or a hybrid solver (see Section 6.3).

The G12 Dantzig-Wolfe decomposition and column generation solver interface implements most of the standard functionality of the G12 LP solver interface. From the outside it looks mostly like a standard LP solver set up with the original problem using the original (linearised) variables. The mapping between the original variables and the master problem variables is straightforward; we simply set

$$x^k = \sum\nolimits_{p \in P^k} d_p^k \lambda_p^k.$$

The main difference to a standard LP solver lies in the initialisation of the column generation module. First the subproblem solver instances have to be added, then the variables to be decomposed are created, and finally the master problem constraints are posted.

Similarly to the simplex algorithm, column generation requires an initial feasible solution. If none is provided by the user, the column generation module introduces artificial variables in order to determine it automatically. At the end of this first phase the artificial variables are removed from the problem [32].

5.2 Branching

The column generation algorithm only solves the LP-relaxed version of the problem; thus branching is required to ensure integrality of the integer variables. The default G12 branch-and-bound module is a simple, standard linear programming based branch-and-bound algorithm as outlined in Section 3. It branches on the original model variables and does not affect the subproblem structure.

The availability of the original variables in the column generation module is the key to being able to use it in further hybrids. We can use it with an arbitrary search strategy on the original variables, or for example in combination with an FD solver, by communicating bounds on the original variables.

5.3 Computational Experiments

We solved various instances of our trucking example, showing the advantages of using DW decomposition. Table 1 lists the results for five instances. Four solvers were used: the G12 finite domain constraint programming solver (FD) on the original (non-linearised) model, the regular G12 LP-based branch-and-bound module (LP-BB) with CPLEX (version 10.0) as the LP solver, CPLEX as a stand-alone MIP solver (CPX-MIP), and the G12 DW decomposition and column generation module (DW). For the latter, we used the G12 branch-and-bound module with CPLEX as LP solver for the linearised master problem and CPLEX as IP solver for the column generation subproblem. In principle, any kind of LP solver can be used as the master solver, and any kind of subproblem solver is possible, as long as G12 interfaces are provided.

We report the times required for solving the problem, and the number of search nodes for FD and LP-BB. For the examined instances, the DW decomposition is so strong that it yielded the optimal integral solution in the root node without a need to branch; so instead of the number of nodes we show the number of columns generated for the DW decomposed problem. The results of stand-alone CPLEX are comparable to the ones obtained using the column generation module, suggesting that the overhead introduced by the G12 column generation library is not very significant.

Table 1 Results for the trucking problem: constraint programming vs. LP-based branch-and-bound vs. CPLEX vs. DW decomposition.

Instance		FD		LP-BB		CPX-MIP	DW		
Trucks	Periods	Nodes	Time	Nodes	Time	Time	Columns	LP/IP opt.	Time
4	6	4655	0.80s	3282	0.55s	0.12s	19	220.0	0.18s
4	6	5860	0.85s	1992	0.47s	0.16s	12	210.0	0.16s
4	6	4607	0.77s	3102	0.55s	0.16s	20	224.0	0.18s
4	8	39848	5.04s	25646	2.64s	0.16s	24	324.0	0.18s
5	7	2361926	215.90s	194000	18.75s	0.18s	18	287.0	0.18s

6 Advanced Concepts in the G12 Implementation

The basic Dantzig-Wolfe decomposition, column generation, and branch-and-bound scheme defined in the previous section is a good theoretical starting point. In general we are often confronted with problems with special subproblem structures requiring more advanced techniques enabling more efficient solving. Our implementation provides some of those techniques while others, such as column pool management or various stabilisation techniques [21], still remain to be added to our system.

6.1 Identical Subproblems

Dantzig-Wolfe decomposition often results in highly symmetrical models because of structurally identical subproblems, i.e. the objective coefficients, the master problem constraints and the subproblem constraints are identical. A typical example for such a model is the cutting stock problem where all the stock pieces have the same dimensions [19,15].

Solving problems with identical subproblems by the pure Dantzig-Wolfe approach can be quite inefficient. This issue is usually overcome by aggregating the identical subproblems. The set K of subproblem indices is partitioned into sets K^s by grouping the indices of identical subproblems; s ranges over some S. We turn

$$\sum\nolimits_{k\in K^s}\sum\nolimits_{p\in P^k} d_p^k\lambda_p^k \qquad \text{into} \qquad \sum\nolimits_{p\in P^s} d_p^s\lambda_p^s$$

where λ_p^s are integer variables satisfying $0 \le \lambda_p^s \le |K^s|$ and the convexity constraint $\sum_{p\in P^s}\lambda_p^s = |K^s|$.

The Master Problem MP becomes the *Aggregated Master Problem*:

$$\begin{aligned}
\textsf{AMP}: \quad \text{minimise} \quad & \sum\nolimits_{s\in S}\sum\nolimits_{p\in P^s} c^s d_p^s \lambda_p^s & \\
\text{subject to} \quad & \sum\nolimits_{s\in S}\sum\nolimits_{p\in P^s} A_j^s d_p^s \lambda_p^s \ge b_j & \forall j = 1\ldots M \\
& \sum\nolimits_{p\in P^s} \lambda_p^s = |K^s| & s \in S \\
& \lambda_p^s \le |K^s|,\ \lambda_p^s \in \mathbb{Z}_+ & \forall p \in P^s,\ s \in S.
\end{aligned}$$

6.2 Automatic Disaggregation when Branching on Original Variables

The direct mapping between the original variables and the newly introduced variables is not obvious anymore. In the aggregated case we have

$$x^k = \sum\nolimits_{p \in P^s} \lambda_p^s d_p^s / |K^s|.$$

This usually leads to highly fractional values for the original variables, even if the λ_p^s variables take integer values. Integrality is preserved as much as possible by greedily decomposing the λ_p^s values into their non-aggregated counterparts (λ_p^k), assigning a non-integral value to at most two of the λ_p^k. First, λ_p^s is decomposed into a fractional (f) and an integral (i) part. Second, it is made sure that decomposed convexity constraints are not violated

$$\sum\nolimits_{p \in P^k} \lambda_p^k = 1 \qquad \forall k \in K^s.$$

This is achieved by assigning the fractional part to the first $\lambda_p^q, q \in K^s$ where $\sum_{p \in P^q} \lambda_p^q < 1$. If $\sum_{p \in P^q} \lambda_p^q + f > 1$, the remaining fractional part is assigned to $\lambda_p^r, r \in K^s$ with $\sum_{p \in P^r} \lambda_p^r = 0$. If after this step $\sum_{p \in P^q} \lambda_p^q = 1$, then q is removed from K^s. The integral part i is then distributed onto $\lambda_p^l, l \in K^s$ where $\sum_{p \in P^l} \lambda_p^l = 0$. These indices are removed from K^s as well, thus ensuring that the decomposed convexity constraints are not violated. Finally, the mapping for the non-aggregated case is applied.

In order to allow branching on the original variables we have to disaggregate the problem as required by the branching. As explained earlier, the column generation module allows one to post any kind of linear constraint on the original problem variables without affecting the subproblem structure. Each aggregated subproblem appearing in these constraints is automatically disaggregated and separately considered by the column generation iterations in the subsequent nodes. Given K identical subproblems, if a constraint is posted involving an original variable belonging to the kth subproblem, this subproblem becomes different from the others and is disaggregated (while the remaining $K - 1$ subproblems are kept aggregated). In order to implement this complex behaviour, the column generation module maintains a mapping between the original variables and their associated subproblems. It also tracks the aggregation status of all the subproblems by keeping a list of active subproblems. The disaggregations are automatically rolled back upon backtracking.

The Cutting Stock problem. In this classic problem, we are given `N` items with associated lengths and demands. We are further given stock pieces with length `L` and an upper bound `K` on the number of required stock pieces for satisfying the demand (a trivial upper bound is the sum over all the demands).

The Zinc model of this problem contains subproblem annotations as seen before, but it also makes use of new annotations. The first one,

colgen_symmetric, annotates a type. It indicates that the model is symmetric in this dimension and that the resulting column generation should aggregate in it.

The second, ***colgen_ph(ann: solver, int: nnodes, int: time)***, adds a primal heuristic to the standard LP based branch-and-bound solver. Every *nnodes* nodes the solver *solver* is called for *time* seconds on the master problem to possibly derive a feasible (integral) solution.

The following Zinc model corresponds to the formulation by Kantorovich [19]:

CuttingStock.zinc - user defined

```
int: K;                           type Pieces = 1..K :: colgen_symmetric;
int: N;                           type Items  = 1..N;
int: L;
array[Items] of int: i_length;
array[Items] of int: i_demand;

array[Pieces]        of var 0..1: pieces;
array[Pieces, Items] of var int:  items;

constraint
    forall(i in Items)(
        sum([ items[k,i] | k in Pieces ]) >= i_demand[i]
    );

constraint
    forall( k in Pieces)(
        (sum(i in Items)( items[k,i] * i_length[i] ) <= pieces[k] * L)
            :: colgen_subproblem_constraint(k, knapsack)
    );

solve :: colgen_solver(lp)
      :: colgen_ph(mip, 100, 10)
      :: lp_bb(pieces ++ [ items[k,i] | k in Pieces, i in Items ],
            most_frac, std_split)
    minimize sum([ pieces[k] | k in Pieces ]);
```

The Cadmium column generation reformulation (see Section 4) recognises the symmetry annotation. It processes the model similarly to the non-symmetric case but it projects out the array component corresponding to the symmetric index in subproblem constraints. For the cutting stock example, the following aggregated version of the variables and constraints is produced:

CuttingStock_2.zinc - automatically generated

```
...

var 0..1: s_pieces :: colgen_subproblem_var(knapsack);
var int:  m_pieces :: colgen_master_var;

array[Items] of var int: s_items :: colgen_subproblem_var(knapsack);
array[Items] of var int: m_items :: colgen_master_var;

constraint
    colgen_link(pieces, m_pieces, s_pieces);

constraint
    forall(i in Items)(
        colgen_link([ items[k,i] | k in Pieces ], m_items[i], s_items[i])
    );
```

```
constraint
    forall(i in Items)( m_items[i] >= i_demand[i] );

constraint
    (sum(i in Items)( s_items[i] * i_length[i] ) <= s_pieces * L)
        :: colgen_subproblem_constraint(0, knapsack);

solve :: colgen_solver(lp)
      :: colgen_ph(mip, 100, 10)
      :: lp_bb(pieces ++ [ items[k, i] | k in Pieces, i in Items ],
            most_frac, std_split)
    minimize m_pieces;
```

As in the non-symmetric case, the `colgen_link` constraints associate the aggregated master and subproblem variables with the original problem variables. The `m_pieces` and `m_items` variables are place-holders representing the implicit sums of aggregated λ variables $\sum_{p \in P^s} d_p^s \lambda_p^s$ as introduced in the AMP. The `s_pieces` and `s_items` variables are the actual subproblem variables.

This model is similar to the well-known column generation formulation first described by Gilmore and Gomory [15], although that does not retain the original variables. Note that it is quite conceivable to use Cadmium to *detect* symmetries and automatically add *colgen_symmetric* annotations.

We experimentally evaluated possible differences when using the aggregated and the non-aggregated DW decomposition. The results are shown in Table 2. We display in percent how often a proven optimal solution or a feasible solution was found. We further give average objective values where at least a feasible (but not proven optimal) solution was found. Average run-times over all the instances are also shown. The maximum run-time per instance was 5 minutes. We used CPLEX as LP solver and a specialised dynamic programming algorithm implemented in Mercury for solving the knapsack subproblems. The CPLEX MIP solver was used as primal heuristic to solve the restricted master problem to integrality at every 100th node with a time limit of 10 seconds, as specified by the *colgen_ph* annotation. The instances were randomly generated using CUTGEN1 [14]. Instances of Classes 1–12 have stock length $L = 1000$; each class consists of 10 instances. We further display the results obtained by applying the CPLEX MIP solver to the original problem.

For almost all classes, aggregating identical subproblems presents an advantage in the number of solved instances, solution quality and solving time. We anticipated these results, since aggregation can strongly increase the number of explored nodes in the limited amount of time given.

6.3 Hybrid LP-Solving of the Master Problem

In column generation a near-optimal solution is usually reached relatively quickly, but the closer to the optimum one gets, the smaller the progress per iteration becomes. Sometimes the speed and convergence behaviour of

Table 2 Results for cutting-stock with a maximum run-time of 5 min.

Class	Items	Original model				No aggregation				Aggregation			
		Opt %	Feas %	Obj	Time [s]	Opt %	Feas %	Obj	Time [s]	Opt %	Feas %	Obj	Time[s]
Class1	10	90	10	11.70	10	30	70	12.70	210	30	70	12.60	210
Class2	10	20	80	114.00	241	70	10	118.75	101	90	10	112.90	59
Class3	20	100	0	21.30	< 1	30	0	23.33	243	20	80	24.50	250
Class4	20	0	100	210.90	300	0	0	n.a.	299	10	30	222.50	268
Class5	10	10	90	49.60	270	100	0	49.50	6	100	0	49.50	< 1
Class6	10	10	90	495.80	273	80	10	518.56	68	100	0	494.90	22
Class7	20	30	70	90.10	211	70	20	90.22	105	90	10	90.00	50
Class8	20	0	100	902.40	340	60	0	947.83	184	90	10	893.50	31
Class9	10	40	60	64.00	187	100	0	64.00	2	100	0	64.00	2
Class10	10	20	80	639.60	230	80	10	657.67	70	90	10	639.70	39
Class11	20	10	90	115.20	270	70	10	117.75	95	80	20	115.50	60
Class12	20	0	100	1149.50	337	70	10	1182.25	155	80	20	1146.90	50

column generation can be improved using non-optimal dual solutions, for example generated by the volume algorithm [3, 2, 21].

Because of the incompleteness of the volume algorithm, we implemented a hybrid solver. It first solves the LP using the volume algorithm and, if called a second time on an unchanged problem, solves it using a standard simplex solver. We adapted the column generation algorithm in such a way that the LP solver for the master problem is called a second time on the unchanged master problem, if the first call to solve did not guarantee an optimal dual solution and no variable with negative reduced cost was found. If a new column with negative reduced cost was found using the optimal dual solution of the standard LP solver it is added to the master problem, and the hybrid implementation switches back to the volume algorithm.

We added the possibility to choose such a hybrid volume algorithm/LP solver for the master problem. It can be selected by a ***lin_hyb(volume, lp)*** annotation on the solve goal; for example, in the cutting-stock problem:

```
solve :: lin_hyb(volume, lp)
      :: colgen_ph(mip, 100, 10)
      :: lp_bb(pieces ++ [ items[k, i] | k in Pieces, i in Items ],
            most_frac, std_split)
   minimize pieces;
```

In Table 3 we present the cutting-stock experiments from the previous section but using the hybrid algorithm instead of the pure LP master solver. The hybrid improves the number of instances solved to optimality and the required run-times. These improvements are mainly due to the fact that the columns generated are better than the ones generated by the pure LP master solver, since the primal heuristic is capable of finding an optimal or near-optimal solution earlier.

6.4 Specialised Branching Rules

In order to overcome symmetry issues, specialised branching rules for specific problem types were developed; see e.g. [4]. They usually require changes to the subproblems during the branch-and-bound process. G12 enables users

Table 3 Results for cutting-stock with a maximum run-time of 5 min. using the LP/volume algorithm hybrid master solver.

Class	Items	Original model				Aggregation				Aggregation/volume			
		Opt %	Feas %	Obj	Time [s]	Opt %	Feas %	Obj	Time [s]	Opt %	Feas %	Obj	Time[s]
Class1	10	90	10	11.70	10	30	70	12.60	210	80	20	12.10	64
Class2	10	20	80	114.00	241	90	10	112.90	59	90	10	112.90	30
Class3	20	100	0	21.30	< 1	20	80	24.50	250	10	90	24.80	270
Class4	20	0	100	210.90	300	10	30	222.50	268	20	20	211.25	240
Class5	10	10	90	49.60	270	100	0	49.50	< 1	100	0	49.50	< 1
Class6	10	10	90	495.80	273	100	0	494.90	22	100	0	494.90	< 1
Class7	20	30	70	90.10	211	90	10	90.00	50	100	0	89.90	1
Class8	20	0	100	902.40	340	90	10	893.50	31	90	10	893.50	30
Class9	10	40	60	64.00	187	100	0	64.00	2	100	0	64.00	2
Class10	10	20	80	639.60	230	90	10	639.70	39	90	10	639.70	30
Class11	20	10	90	115.20	270	80	20	115.50	60	80	20	115.50	60
Class12	20	0	100	1149.50	337	80	20	1146.90	50	80	20	1146.90	67

to implement such specialised branching rules, changing the structure of the subproblems but preserving aggregations.

The column generation module allows one to ask for fractional columns of the DW decomposed model. It returns their values as well as their entries in the constraint matrix of the master problem. Using this information the user can define specialised branching rules by introducing constraint branches on subproblem variables. In the master problem these constraint branches can be enforced by setting forbidden columns to zero in their respective branch. The column generation module provides a predicate by which the user can specify a list of column patterns that have to be set to zero. In our current system the specialised branching rules are implemented in Mercury and can be accessed through model annotations.

The Two-Dimensional Bin Packing problem. In order to demonstrate the effectiveness of specialised branching rules, we implemented a simple, well-known rule for the two-dimensional bin packing problem.

We are given `N` rectangular items of given height and width. These items have to be placed on (or cut out) of bins of height `H` and width `W`, of which there are at most `K`. Item rotations are not allowed and only level packings (or 2-stage cuttings) are feasible. Each bin can be divided into several levels, and each level contains the items next to each other [20]. For ease of modelling, we assume that the items are sorted by non-increasing heights.

The annotation ***bp(array[int] var int: x, ann: choice, ann: split)*** determines the use of the branch-and-price algorithm with branching on subproblems. The formulation in Zinc is as follows:

2DBinPacking.zinc

```
int: K;                          type Bins  = 1..K :: colgen_symmetric;
int: N;                          type Items = 1..N;
int: W;
int: H;

array[Items] of int: ItemWidth;
array[Items] of int: ItemHeight;
```

```
array[Bins]        of var 0..1: bin;
array[Bins, Items] of var 0..1: item;

constraint
    forall(j in Items)( sum(k in Bins)( item[k, j] ) >= 1 );

constraint
    forall(k in Bins)(
        is_feasible_packing(bin[k], [ item[k, j] | j in Items ])
            :: colgen_subproblem_constraint(k, mip)
    );

set of tuple(Items, Items): Idx = { (i, j) | i, j in Items where j >= i };

predicate
    is_feasible_packing(var 0..1: l_bin, array[Items] of var 0..1: l_item) =
        let { array[Idx] of var 0..1: x }
        in
        forall (i in Items)(
            sum(j in Items)( ItemWidth[j] * x[i, j]) <= W * x[i, i]
        )
        /\
        sum(i in Items)( ItemHeight[i] * x[i, i] ) <= l_bin * H
        /\
        forall(j in Items)( l_item[j] = sum(i in 1..j)( x[i, j] ) );

solve :: colgen_solver(lp)
      :: colgen_ph(mip, 100, 10)
      :: bp(bin ++ [ item[k,j] | k in Bins, j in Items ],
            most_frac_master, special_split)
    minimize sum(k in Bins)( bin[k] );
```

The *bp* annotation tells the branch-and-price algorithm to use the most fractional master variable choice and the specialised branching rule. The specialised branching rule corresponds to the one described in [24], which is based on a well known branching rule for set partitioning [27]. The solution space is divided by branching on whether two different items are in the same bin. We always choose the two highest items `u` and `v` appearing in a pattern whose corresponding column generation master variable λ has an LP solution value closest to 0.5. The branches are created by adding

```
sum(i in 1..u)(x[i, u]) + sum(i in 1..v)(x[i, v]) <= 1
```

to the subproblems in the first branch and

```
sum(i in 1..u)(x[i, u]) = sum(i in 1..v)(x[i, v])
```

to the subproblems in the second branch. The branching has to be further enforced in the master problem, which is done by deleting those columns that violate the newly added subproblem constraints. The Mercury column generation module supports the implementation of special (subproblem based) branching rules by providing functionality for deleting columns according to column patterns specified by the branching implementation.

The modelling of the problem in Zinc and the implementation of the specialised branching in Mercury was a matter of days. One of the authors has implemented a customised branch-and-price algorithm for 3-stage two-dimensional bin packing [24] using the COIN-OR/Bcp library requiring substantially more effort (a few months). The implementation required

Table 4 Results for two-dimensional bin packing with a maximum run-time of 5 min.

Class	Original model				Standard CG				CG with sp. branching			
	Opt %	Feas %	Obj	Time [s]	Opt %	Feas %	Obj	Time[s]	Opt %	Feas %	Obj	Time[s]
Class1	28	56	19.88	229	65	7	19.79	138	86	4	19.33	84
Class2	48	52	5.32	189	28	0	1.36	224	24	0	1.17	226
Class3	26	48	14.68	245	54	6	11.84	160	70	2	12.22	137
Class4	50	50	3.84	178	20	0	1.00	232	20	0	1.00	232
Class5	28	60	18.80	237	75	6	17.07	100	82	6	17.07	81
Class6	54	46	3.26	180	22	0	1.00	227	22	0	1.00	226
Class7	44	54	18.31	190	56	16	17.33	141	86	14	16.80	71
Class8	20	80	18.64	267	62	12	16.68	141	80	10	16.47	104
Class9	58	26	38.83	163	96	0	41.50	22	96	0	41.62	32
Class10	26	74	12.64	257	42	4	7.46	189	50	4	7.70	175

the modelling of the problem, reading of instance data, initialisation of the master problem using feasible solutions, implementation of the branching rules and dealing with resulting infeasible branches (restoring feasibility), implementation of the subproblem solving procedure and many other aspects.

Table 4 displays the results of applying standard branching on the original variables or using the specialised branching rule. We tested these approaches on the set of 500 instances described in [20]. They are divided in 10 classes of 50 instances each, with item numbers ranging from 20 to 100 in each class. While many instances could be solved to optimality in the root node, the specialised branching rules did reach optimal solutions more often in the given limited run-time. The number of problems without solutions could also be reduced while requiring lower average run-times. We also applied CPLEX MIP to the original model. One can observe that this yields significantly fewer proven optimal solutions in the limited run-time, while it allows CPLEX to find more feasible solutions in total.

7 Conclusion

As noted in Section 2 there are other systems that allow the same model to run using different solvers or solving approaches. However, we are not aware of any system, other than G12, that supports the complex rewriting required to take a solver-independent model and solve it using column generation. The second novelty is the ability to perform search on user variables such that any symmetries which are dynamically broken during search are still correctly, efficiently and automatically handled by the column generation solver. Thirdly, the facility to define specialised search still using the mapping managed by the library provides the full flexibility needed by the expert user.

The G12 scheme is to add high-level annotations to a conceptual problem model, which allows it to be turned into a design model that maps to a specific algorithm. Annotating a constraint in the conceptual model with a solver that will handle it is a simple example of this scheme.

Column generation is an interesting challenge because it does not naturally fit into the above scheme. Certainly we view the column generation module as a solver in the normal way. However annotating a constraint with the column generation solver is not enough: the solver needs to know which problem component the constraint belongs to, the master problem or the subproblem. Moreover there is not one column generation solver: the master problem might be sent to one underlying solver and the subproblem to another. Our system also allows the use of hybrid master solvers such as the volume algorithm based master we describe in Section 6.3. Finally branch-and-price search is closely connected with the column generation solver, and annotations to control the search can be crucial to the performance of the algorithm.

Each requirement has been satisfied in Zinc by having a sufficiently expressive annotation language. For example an annotation with a compound term, *colgen_subproblem_constraint(p, mip)*, was used to specify the subproblem solver in Section 4, and the search was specified by multiple annotations.

The next particular challenge of column generation is that the variables (and constraints) used in the conceptual model of the problem are quite different from those needed in the design model. Our column generation module automates this mapping using G12's Cadmium mapping language. To ensure the annotations are still meaningful with respect to the new variables, the annotations have to be transformed by Cadmium in the same way. Moreover the search control as illustrated in Section 6.2 must be mapped to search steps expressed in terms of the design model variables.

The greatest design and implementation challenge was to have these still work, fully automatically, when handling symmetry by generating aggregated variables (used when solving the subproblem) and dynamically disaggregating some of them during search. Indeed, each symmetry-breaking search step causes the design model to be updated so as to operate on a new set of variables.

One interesting challenge arising out of this work is how to automatically detect identical subproblems. This is a completely novel form of automated symmetry detection, which is of significant practical value, as the results in Table 2 reveal. Finally, we envisage to explore the use of the column generation module for solving a subproblem within a larger problem – thus supporting, for example, a combination of row and column generation.

Acknowledgements

We would like to thank the members of the G12 team at NICTA VRL for helpful discussions and implementation work.

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.
2. R. Anbil, J. Forrest, and W. Pulleyblank. Column generation and the airline crew pairing problem. In *Documenta Mathematica, Extra Volume ICM*, 1998.
3. F. Barahona and R. Anbil. The volume algorithm: producing primal solutions with a subgradient method. *Mathematical Programming*, 87(3):385–399, 2000.
4. C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Operations Research*, 46(3):316–329, 1998.
5. N. Boland and T. Surendonk. A column generation approach to delivery planning over time with inhomogeneous service providers and service interval constraints. *Annals of Operations Research*, 108:143–156, 2001.
6. S. Brand, G. J. Duck, J. Puchinger, and P. J. Stuckey. Flexible, rule-based constraint model linearisation. In P. Hudak and D. Warren, editors, *Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 68–83. Springer, 2008.
7. A. Chabrier. *Génération de Colonnes et de Coupes utilisant des sous-problèmes de plus court chemin.* PhD thesis, Université d'Angers, France, 2002.
8. G. B. Dantzig and P. Wolfe. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960.
9. G. Desaulniers, J. Desrosiers, and M. Solomon, editors. *Column Generation.* GERAD 25th Anniversary Series. Springer, 2005.
10. G. J. Duck, P. J. Stuckey, and S. Brand. ACD term rewriting. In S. Etalle and M. Truszczynski, editors, *Logic Programming (ICLP 2006)*, volume 4079 of *LNCS*, pages 117–131. Springer, 2006.
11. ECL^{i}PS^e. www.eclipse-clp.org, 2009.
12. A. Eremin. *Using Dual Values to Integrate Row and Column Generation into Constraint Logic Programming.* PhD thesis, Imperial College London, 2003.
13. M. J. Garcia de la Banda, K. Marriott, R. Rafeh, and M. Wallace. The modelling language Zinc. In F. Benhamou, editor, *Principles and Practice of Constraint Programming (CP'06)*, volume 4204 of *LNCS*, pages 700–705. Springer, 2006.
14. T. Gau and G. Wäscher. CUTGEN1: a problem generator for the standard one-dimensional cutting stock problem. *European Journal of Operational Research*, 84(3):572–579, 1995.
15. P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem (part I). *Operations Research*, 9:849–859, 1961.
16. O. Gunluk, L. Ladanyi, and S. D. Vries. A branch-and-price algorithm and new test problems for spectrum auctions. *Management Science*, 51(3):391–406, 2005.
17. M. Jünger and S. Thienel. The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Software: Practice and Experience*, 30(11):1325–1352, 2000.
18. U. Junker, S. E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. A framework for constraint programming based column generation. In J. Jaffar, editor, *Principles and Practice of Constraint Programming (CP'99)*, volume 1713 of *LNCS*, pages 261–274. Springer, 1999.

19. L. V. Kantorovich. Mathematical methods of organizing and planning production. *Management Science*, 6(4):366–422, 1960.
20. A. Lodi, S. Martello, and D. Vigo. Models and bounds for two-dimensional level packing problems. *Journal of Combinatorial Optimization*, 8(3):363–379, 2004.
21. M. Lübbecke and J. Desrosiers. Selected topics in column generation. *Operations Research*, 53(6):1007–1023, 2005.
22. G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.
23. N. Papadakos. Integrated airline scheduling. *Computers and Operations Research*, 36:176–195, 2009. To appear; available online 27 August 2007.
24. J. Puchinger and G. R. Raidl. Models and algorithms for three-stage two-dimensional bin packing. *European Journal of Operational Research*, 183(3):1304–1327, 2007.
25. T. Ralphs and L. Ladanyi. COIN/BCP user's manual, 2001.
26. L.-M. Rousseau, M. Gendreau, G. Pesant, and F. Focacci. Solving VRPTWs with constraint programming based column generation. *Annals of Operations Research*, 130(1):199–216, 2004.
27. D. M. Ryan and B. Foster. An integer programming approach to scheduling. In A. Wren, editor, *Computer scheduling of public transport urban passenger vehicle and crew scheduling*, pages 269–280. North Holland, Amsterdam, 1981.
28. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
29. P. J. Stuckey, M. J. G. de la Banda, M. J. Maher, K. Marriott, J. K. Slaney, Z. Somogyi, M. Wallace, and T. Walsh. The G12 project: Mapping solver independent models to efficient solutions. In P. van Beek, editor, *Principles and Practice of Constraint Programming (CP'05)*, volume 3709 of *LNCS*, pages 13–16. Springer, 2005.
30. P. Van Hentenryck and L. Michel. OPL Script: Composing and controlling models. In K. R. Apt, A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints*, volume 1865 of *LNCS*, pages 75–90. Springer, 1999.
31. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. The MIT Press, 2005.
32. F. Vanderbeck. Branching in branch-and-price: a generic scheme. Technical Report U-05.14, Applied Mathematics, University Bordeaux 1, France, 2005.
33. D. Villeneuve, J. Desrosiers, M. E. Lübbecke, and F. Soumis. On compact formulations for integer programs solved by column generation. *Annals of Operations Research*, 139(1):375–388, 2005.
34. T. Yunes, I. Aron, and J. Hooker. An integrated solver for optimization problems (updated on 6/10/09). Technical report, University of Miami, 2009.
35. T. H. Yunes, A. V. Moura, and C. C. de Souza. A hybrid approach for solving large scale crew scheduling problems. In *Practical Aspects of Declarative Languages (PADL'00)*, volume 1753 of *LNCS*, pages 293–207. Springer, 2000.