# From High-Level Model to Branch-and-Price Solution in G12

Jakob Puchinger, Peter J Stuckey, Mark Wallace, Sebastian Brand

# From High-Level Model to Branch-and-Price Solution in G12

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

[1] NICTA Victoria Research Laboratory
Department of Computer Science & Software Engineering
University of Melbourne, Australia
{jakobp,pjs,sbrand}@csse.unimelb.edu.au
[2] School of Computer Science and Software Engineering
Monash University, Melbourne, Australia
mgw@mail.csse.monash.edu.au

**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. G12 supports the selection of specialised sub-problem solvers, the aggregation of identical sub-problems, automatic disaggregation when required by search, and the use of specialised branching rules. We demonstrate the benefits of the G12 framework on three examples: a trucking problem, cutting stock, and two-dimensional bin packing.

## 1 Introduction

Combinatorial optimisation problems are easy to state, but hard to solve, and 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 [27]. 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. 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 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.

The G12 platform consists of three major components, the modelling language Zinc [12], the model transformation language Cadmium [10], and several internal and external solvers written and/or interfaced using the general-purpose programming language Mercury [26]. All solvers and solver instances are specified in terms of their specific capabilities, i.e. the type of problems they can solve, the type of information they can return, and how they solve a problem.

On the Mercury level these specifications are described using type classes. Basic solvers such as a Finite Domain Constraint Programming (FD) solver or a Linear Programming (LP) solver can be used as building blocks for other solvers. The column generation and branch-and-price modules use and implement such solver type classes. This system of pluggable components allows us to quickly design new hybrid algorithms and to combine existing solvers in innovative ways.

**Trucking problem.** Consider the following trucking 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 `T` 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 follows:

Trucking.zinc

```
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. 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", "indomain", "complete")
      minimize sum(p in Periods)(sum_set(x[p], Loads) >= Demand[p]);
```

which indicates we label the set variables with smallest domain first (`first_fail`) by first trying to exclude an unknown element of the set and then including it (`indomain`) in a complete search.

## 2 Dantzig-Wolfe Decomposition and Column Generation

Dantzig-Wolfe decomposition is a standard way to decompose an integer programming model into a master problem and one or several subproblems [8, 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.

The *Original Problem* has the form

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

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

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

Substituting the $x^k$ by the $\lambda_p^k$ in $\mathsf{OP}$, we obtain the *Master Problem*:

$$\begin{array}{lllll} \mathsf{MP}: & \text{minimise} & \sum_{k \in K} \sum_{p \in P^k} c^k d_p^k \lambda_p^k & & \\ & \text{subject to} & \sum_{k \in K} \sum_{p \in P^k} A_j^k d_p^k \lambda_p^k \geq b_j & \forall j = 1 \ldots M & (1) \\ & & \sum_{p \in P^k} \lambda_p^k = 1 & k \in K & (2) \\ & & \lambda_p^k \in \{0,1\} & \forall p \in P^k,\ k \in K. & \end{array}$$

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 (RMP), variables (columns) are lazily 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 profitable 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 determine such variables. In case of a minimisation problem, the objective is to find feasible columns $d^k$ with negative reduced cost:

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

where $\pi$ are the dual variable values corresponding to the constraints (1) and $\mu^k$ is the dual value of the $k$th convexity constraint (2). We do not need to find a column with maximum profit; adding a "good" column is sufficient.

## 3 Solving with G12

The G12 system allows one to take a model written in Zinc, transform it to various underlying solvers using Cadmium, and then execute it. We can use standard or user-defined Cadmium transformations. Mappings from Zinc to FD or LP models are available [6]. To control these transformations the user can annotate the model. The trucking problem example illustrates the use of an annotation to define search for an FD solver.

At the solver programming language (Mercury) level, G12 defines interfaces to solvers such as an FD solver, a continuous interval constraint solver, and linear programming solvers using type classes. Various implementations of these interfaces are provided, e.g. for LP/MIP solvers such as CPLEX, COIN-OR/OSI, and others. 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.

The advantage of this architecture is that we can easily plug different LP solvers into modules such as column generation and branch-and-bound.

### 3.1 Dantzig-Wolfe Decomposition and Column Generation in Zinc

In order to use Dantzig-Wolfe decomposition and column generation on a high-level model in G12, we need to annotate the model to explain: what parts define the sub-problems, which solver is to be used for each subproblem, and which solver is to be used for the master problem.

For instance, the trucking problem example can be annotated as follows:

```
array[Periods] of var set of Trucks: x :: colgen_var;

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));
```

which exposes which variables `x` will be used in column generation. 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. Finally, the solver for the master problem and the search specification, branch-and-bound selecting the most fractional variable first and performing a standard split, are attached to the `solve` item.

We then perform a Dantzig-Wolfe decomposition on the model, separating original, master, and subproblem variables, as well as adding constraints linking those variables:

Trucking.zinc (changes)

```
array[Periods] of var set of Trucks: mx :: colgen_master_var;
array[Periods] of var set of Trucks: sx :: colgen_subproblem_var;

constraint forall(p in Periods)(
    colgen_link_constraint([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));
```

The so called master variables are place-holders representing the implicit sums of the $\lambda$ variables $\sum_{p \in P^k} d_p^k \lambda_p^k$ as introduced in MP. Note that the search is still expressed in terms of the original problem variables.

Since column generation is to be used, the transformation must linearise the master constraints and objective function. The subproblem solver could use the original set representation of the variables, but for this example it too requires linearisation of the sub-problem constraints.

We can linearise the master and subproblem constraints giving linear definitions for the `sum_set` and `sequence` globals. This can be done in Zinc as:

```
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);
```

where `index_set` returns the set of indices of its array argument, and `bool2int` coerces a Boolean to `0..1`. Finally, we transform the array of set variables `x` to a two-dimensional array of `0..1` variables such that `x[p,t] = 1` if `t in x[p]`.

### 3.2 Implementation

Column generation works by first transforming the original model to the form demanded by the solvers. Then 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, although we are working on adding a hybrid volume algorithm/LP solver [3, 2].

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 straight-forward; we simply set

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

The main difference 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 it is not provided by the user, we introduce artificial variables in order to determine it automatically. At the end of this first phase the artificial variables are removed from the problem [29].

Since the column generation algorithm alone only solves the LP-relaxed version of the problem, we have to branch in order to guarantee integrality of the variables. The default branch-and-bound module is a simple, standard linear programming based branch-and-bound algorithm branching on the original model variables, which does not affect the subproblem structure [30].

The additional branching constraints could of course render the RMP infeasible. But, since we are usually not dealing with the complete master problem, additional columns could restore feasibility of the RMP. Such columns are obtained by solving a problem very similar to the pricing problem [15].

The availability of the original variables in the column generation solver is the key to being able to use this solver 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.

### 3.3 The Trucking Problem

We solved several different instances of our trucking example showing the advantages of using DW-decomposition. Table 1 shows results on five different instances displaying the number of search nodes and the time required for solving the model using an FD solver, using a linearised model with branch-and-bound (LP-BB), and using DW-decomposition and column generation (DW). For the trucking example the DW-decomposition is so strong that it yielded the optimal

**Table 1.** The trucking example: finite domain model versus linearised branch-and-bound versus DW-decomposition

| Instance | | FD | | LP-BB | | | DW | | |
|---|---|---|---|---|---|---|---|---|---|
| Trucks | Periods | Nodes | Time | Nodes | LP opt. | Time | Columns | LP/IP opt. | Time |
| 4 | 6 | 4655 | 0.80s | 3282 | 177.0 | 0.55s | 19 | 220.0 | 0.18s |
| 4 | 6 | 5860 | 0.85s | 1992 | 177.0 | 0.47s | 12 | 210.0 | 0.16s |
| 4 | 6 | 4607 | 0.77s | 3102 | 177.0 | 0.55s | 20 | 224.0 | 0.18s |
| 4 | 8 | 39848 | 5.04s | 25646 | 267.0 | 2.64s | 24 | 324.0 | 0.18s |
| 6 | 7 | 2361926 | 215.90s | 194000 | 244.8 | 18.75s | 18 | 287.0 | 0.18s |

integral solution in the root node without a need to branch; so instead of nodes we show the number of columns generated for the DW-decomposed problem. We also display the value of the LP-relaxation at the root node for the linear models. For this problem we used our own branch-and-bound module using CPLEX as LP solver as well as IP subproblem solver. In general, any kind of LP solver (with G12 interfaces) can be used as master solver, and also any kind of subproblem solver is supported.

## 4 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 [18, 14].

### 4.1 Aggregating Identical Subproblems

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

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

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

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

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

### 4.2 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

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

Unfortunately, this usually leads to highly fractional values for the original variables, even if the $\lambda_p^s$ variables take integer values. We therefore first decompose the $\lambda_p^s$ values into (non-aggregated) $\lambda_p^k$ values preserving integrality as much as possible, and then we use the mapping for the non-aggregated case.

In order to allow branching on the original variables we have to disaggregate the problem as required by the branching. 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 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 $k$th subproblem, this subproblem becomes different to 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 rolled back upon backtracking.

### 4.3 The Cutting Stock Problem

In the cutting stock 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 following ZINC model corresponds to the formulation by Kantorovich [18]:

CuttingStock.zinc

```
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 :: colgen_var;
array[Pieces, Items] of var int:  items  :: colgen_var;

solve :: lp_bb([pieces, items], most_frac, std_split)
      :: colgen_ph(100, 10) :: colgen_solver("lp")
    minimize sum([ pieces[k] | k in 1..K]);

constraint forall(i in 1..N)(sum([items[k, i] | k in 1..K]) >= i_demand[i]);

constraint forall( k in 1..K)(
               sum(i in 1..N)(items[k,i] * i_length[i]) <= pieces[k] * L
                   :: colgen_subproblem_constraint(k, "knapsack"));
```

The original model is a linear program. The annotations for the column generation variables and subproblems are as before. But this time we introduce a new annotation `colgen_symmetric` which annotates a type. This indicates that the model is symmetric in this dimension and the resulting column generation should aggregate in this dimension. A CADMIUM transformation can then be used to create an aggregated version of the variables and constraints as follows:

CuttingStockAgg.zinc (changes)

```
var 0..1: s_pieces                  :: colgen_subproblem_var;
array[Items] of var int: s_items    :: colgen_subproblem_var;
var int: m_pieces                   :: colgen_master_var;
array[Items] of var int: m_items    :: colgen_master_var;

solve :: lp_bb([pieces, items], most_frac, std_split)
      :: colgen_ph("mip", 100, 10) :: colgen_solver("lp")
    minimize m_pieces;

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 1..N)(m_items[i] >= i_demand[i]);

constraint sum(i in 1..N)(s_items[i] * i_length[i]) <= s_pieces * L
              :: colgen_subproblem_constraint(1, "knapsack");
```

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 $\lambda$ 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 [14], although that does not retain the original variables. Note that it is conceivable to use CADMIUM to *detect* symmetries and automatically add `colgen_symmetric` annotations.

In the following experiment we evaluate 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 an optimal solution, a feasible solution, or no solution was found. We further give average objective values and number of explored nodes where at least a feasible 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 using the `colgen_ph` annotation. The instances used are randomly generated using CUTGEN1 [13]. Instances of Classes 1–12 have stock length $L = 1000$; each class consists of 10 instances.

For almost all classes, aggregating identical subproblems presents an advantage in the number of solved instances, solution quality and solving time.

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

| Class | Items | No Aggregation | | | | | | Aggregation | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Opt % | Feas % | No % | Obj | Nodes | Time [s] | Opt % | Feas % | No% | Obj | Nodes | Time[s] |
| Class1 | 10 | 30 | 70 | 0 | 12.70 | 3325.80 | 210.40 | 30 | 70 | 0 | 12.60 | 3596.60 | 209.95 |
| Class2 | 10 | 70 | 10 | 20 | 118.75 | 125 | 100.89 | 90 | 10 | 0 | 112.90 | 283.80 | 59.36 |
| Class3 | 20 | 30 | 0 | 70 | 23.33 | 766.67 | 242.52 | 20 | 80 | 0 | 24.50 | 823.10 | 250.05 |
| Class4 | 20 | 0 | 0 | 100 | n.a. | n.a. | 298.63 | 10 | 30 | 60 | 222.50 | 400 | 268.17 |
| Class5 | 10 | 100 | 0 | 0 | 49.50 | 75.20 | 6.07 | 100 | 0 | 0 | 49.50 | 0 | 0.32 |
| Class6 | 10 | 80 | 10 | 10 | 518.56 | 38.89 | 68.39 | 100 | 0 | 0 | 494.90 | 143.40 | 21.84 |
| Class7 | 20 | 70 | 20 | 10 | 90.22 | 212.22 | 105.18 | 90 | 10 | 0 | 90 | 225.90 | 50.00 |
| Class8 | 20 | 60 | 0 | 40 | 947.83 | 16.67 | 184.24 | 90 | 10 | 0 | 893.50 | 40.60 | 30.51 |
| Class9 | 10 | 100 | 0 | 0 | 64 | 20 | 2.04 | 100 | 0 | 0 | 64 | 50 | 1.79 |
| Class10 | 10 | 80 | 10 | 10 | 657.67 | 43.78 | 70.08 | 90 | 10 | 0 | 639.70 | 169 | 39.27 |
| Class11 | 20 | 70 | 10 | 20 | 117.75 | 104.75 | 95.10 | 80 | 20 | 0 | 115.50 | 253.60 | 60.15 |
| Class12 | 20 | 70 | 10 | 20 | 1182.25 | 10.25 | 154.79 | 80 | 20 | 0 | 1146.90 | 120.60 | 50.06 |
| Average | | 63.33 | 11.67 | 25 | 330.74 | 457.60 | 128.19 | 73.33 | 21.67 | 5 | 327.46 | 514.61 | 86.79 |

## 5 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 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. We are working on extensions to Zinc so that users will be able to specify such rules at the modelling level.

### 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. It is similar to the one described in [22], which is based on a well known branching rule for set partitioning [25]. The solution space is divided by branching on whether two different items are in the same bin. We always choose the two highest items appearing in a pattern whose corresponding column generation master variable $\lambda$ has an LP solution value closest to 0.5.

In the two-dimensional bin packing problem (2DBPP), 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`. The

variant we consider here does not allow items to be rotated; only level packings are allowed. Each bin can be divided into several levels, and each level contains the items beside each other [19]. For ease of modelling, we assume that the items are sorted by non-increasing heights. 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;                 array[Items] of int: ItemWidth;
int: H;                 array[Items] of int: ItemHeight;

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

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

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 i..N)(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]));
```

The `bp` annotation to the solve item tells the system to use the branch-and-price algorithm choosing the most fractional master variable and using the specialised branching rule.

Table 3 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 [19]. 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, our specialised branching rules did reach optimal solutions more often in the given limited runtime.

## 6 Related Work and Conclusion

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

**Table 3.** Results for two-dimensional bin packing with a maximum run-time of 5 min.

| Class | Std. Branching | | | | | | Sp. Branching | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Opt % | Feas % | No % | Obj | Nodes | Time [s] | Opt % | Feas % | No% | Obj | Nodes | Time[s] |
| Class1 | 68 | 22 | 10 | 19.49 | 45.87 | 109.90 | 90 | 8 | 2 | 39.90 | 41.14 | 53.54 |
| Class2 | 26 | 0 | 74 | 1.31 | 0 | 223.24 | 30 | 2 | 68 | 64.19 | 6.19 | 203.08 |
| Class3 | 70 | 10 | 20 | 13.05 | 10 | 116.37 | 84 | 8 | 8 | 13.85 | 11.87 | 82.90 |
| Class4 | 26 | 0 | 74 | 1.31 | 0 | 228.76 | 26 | 0 | 74 | 1.31 | 0 | 228.74 |
| Class5 | 84 | 6 | 10 | 17.40 | 8.89 | 69.65 | 90 | 2 | 8 | 17.61 | 3.93 | 53.13 |
| Class6 | 24 | 0 | 76 | 1.08 | 0 | 228.03 | 24 | 0 | 76 | 1.08 | 0 | 227.97 |
| Class7 | 76 | 16 | 8 | 16.30 | 33.70 | 80.52 | 88 | 10 | 2 | 16.78 | 57.18 | 57.52 |
| Class8 | 78 | 10 | 12 | 15.73 | 14.77 | 89.48 | 84 | 6 | 10 | 15.98 | 13.60 | 77.04 |
| Class9 | 96 | 4 | 0 | 42.62 | 6.72 | 13.94 | 100 | 0 | 0 | 42.60 | 0.32 | 2.17 |
| Class10 | 48 | 4 | 48 | 7.46 | 18.15 | 155.95 | 52 | 0 | 48 | 7.46 | 7.54 | 149.39 |
| Average | 59.6 | 7.2 | 33.2 | 17.95 | 17.58 | 131.58 | 66.8 | 3.6 | 29.6 | 23.65 | 18.38 | 113.55 |

can be generated by constraint programming or application-specific algorithms, while the master problem is handled using branch-and-price [17, 31, 24, 22].

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 [16], MINTO [20], OPL script [28], MAESTRO [7], COIN/BCP [23], and SCIP [1] offer facilities to support the implementation of 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.

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]. This system 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 [21].

The facility to annotate the same Zinc model in two different ways, as in the examples above, and thus have the problem solved by the FD solver, or by column generation according to the annotation, is completely novel. Moreover the facility to perform search on user variables and have any symmetries which are dynamically broken during search still correctly and efficiently handled automatically by the column generation solver is also new. Thirdly the facility to

define specialised search still using the mapping provided by the library provides the full flexibility needed by the expert user.

The G12 scheme is to add annotations to a conceptual problem model, and thus turn it into a design model that maps to a specific algorithm. Annotating a constraint, occurring in the conceptual model, with the (name of the) 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 (as discussed in Section 3). However annotating a constraint with the column generation solver is not enough: the solver needs to know which subproblem 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. 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 3.1, 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 4 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.

The design model is expressed in terms of a simplified version of ZINC, illustrated in Section 4.3. The specification of our language for expressing design models is still fluid, and so currently the translation to the MERCURY code – which is very similar, but uses different syntax – is by hand.

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.

We plan to implement the search annotation transformations necessary to enable specialised branching schemes to be expressed in ZINC. We also plan to build in an implementation of the generic branching scheme described in [29]. We

further intend to address issues related to adding multiple columns and column pool management.

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. SCIP - a framework to integrate constraint and mixed integer programming. Technical Report 04-19, Zuse Institute Berlin, 2004. http://www.zib.de/Publications/abstracts/ZR-04-19/.
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. A. Eremin. *Using Dual Values to Integrate Row and Column Generation into Constraint Logic Programming*. PhD thesis, Imperial College London, 2003.
12. 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.

13. 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.
14. P. C. Gilmore and R. E. Gomory. A linear programming approach to the cutting-stock problem (part I). *Operations Research*, 9:849–859, 1961.
15. 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.
16. 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.
17. 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.
18. L. V. Kantorovich. Mathematical methods of organizing and planning production. *Management Science*, 6(4):366–422, 1960.
19. 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.
20. G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.
21. N. Papadakos. Integrated airline scheduling. *Computers and Operations Research*, To appear, 2007. Available online 27 August 2007.
22. 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.
23. T. Ralphs and L. Ladanyi. COIN/BCP user's manual, 2001.
24. 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.
25. 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.
26. 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.
27. 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.
28. 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.
29. F. Vanderbeck. Branching in branch-and-price: a generic scheme. Technical Report U-05.14, Applied Mathematics, University Bordeaux 1, France, 2005.
30. 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.
31. 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.