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

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

arsenal research, Vienna, Austria

NICTA, University of Melbourne, Australia

Monash University, Melbourne, Australia

ISDS-Kolloquium 23. März 2009

Outline

G12

An Example

Dantzig-Wolfe Decomposition

Column Generation

Identical Subproblems

Specialised Branching Rules

Conclusions and Outlook

The G12 Solver Platform

- Software environment for stating and solving combinatorial problems by mapping high-level models to efficient combinations of solving methods.
- We develop user-controlled mappings from a high level model to different solving methods.
- Allowing users to experiment with different mappings.
- These mappings must be
  - easy-to-use and easy-to-change for efficient experimentation with alternative hybrid algorithms.
  - flexible, allowing plug-and-play between different sub-algorithms;
  - efficient, allowing, if necessary, to tightly control the behaviour of the algorithm;

Mapping to Branch-and-Price

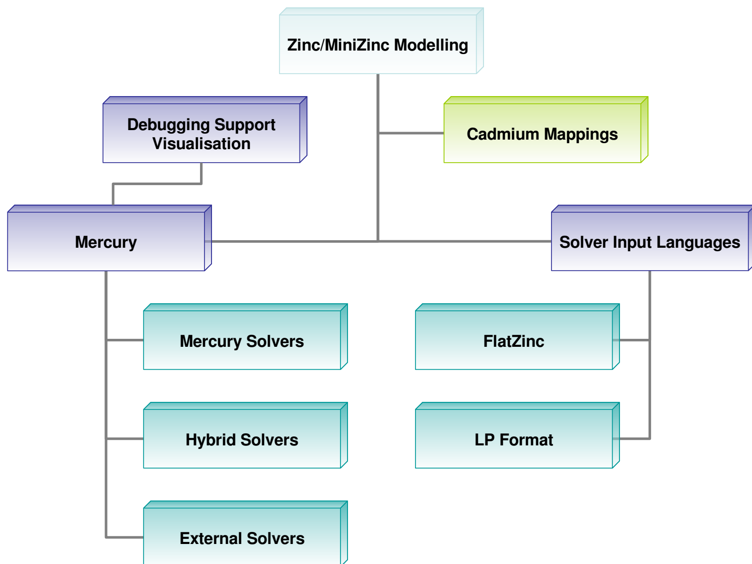
Our mapping to branch-and-price is designed to meet all three objectives:

- 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 separate solvers for the subproblems. The user can control the decomposition and select the subproblem solver.
- Identical subsystems can be aggregated. Search control can be expressed in terms of original model variables. The system also supports specialised branching rules.

Components of the G12 Platform

- ZINC – Modelling Language Family
  - ZINC
  - MINIZINC
  - FLATZINC
- CADMIUM – Mapping Language
- MERCURY – Solver Implementation Language
  System of pluggable components:
  - MERCURY Solvers
  - Hybrid Solvers
  - External Solvers

G12 - Constraint Programming Platform



Example: Trucking Problem

- 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.
- In each consecutive k time periods each truck must be used at least l and at most u times.
- At each time period we need to choose which trucks to use in order to satisfy the constraints.

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) );
```

- The `sum_set`(s, f) function returns $\Sigma_{e \in s} f(e)$.
- `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], Cost));
```

  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.

Dantzig-Wolfe Decomposition

- Decomposing a linear programming model into a master problem and one or several subproblems.
- The *Original Problem* has the form

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

- The $\mathcal{D}^k$ are finite sets of vectors in $\mathbb{Z}_+^{N_k}$ implicitly defined by additional constraints. We have $\mathcal{D}^k = \{d_p^k \mid p \in P^k\}$.

Dantzig-Wolfe Decomposition

- We can alternatively write

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

- Substituting x^k, we obtain the *Master Problem*:

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

Dantzig-Wolfe Decomposition and Column Generation

- DW Decomposition typically results in a Master Problem with a possibly exponential number of variables.
- Column generation is the method of choice:
  - Start from a restricted LP-relaxation of the original problem: the Restricted Master Problem (RMP).
  - Profitable variables (columns) are iteratively included.
  - For every $\mathcal{D}^k$, a subproblem is solved to find such variables.
  - Find feasible columns d^k with negative reduced cost:

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

where π are the dual variable values corresponding to the constraints and μ^k is the dual value of the kth convexity constraint.

Column Generation in G12

In G12 we need to annotate models 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.

Trucking.zinc (changes)

```
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) );
```

Column Generation in G12

Implicit 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: 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) );

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));
```

Implementation

- In the G12 system, the column generation module looks almost like any other LP solver from the outside.
- The mapping between the original variables and the master problem variables is straight-forward; we simply set

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

- Colgen requires an initial feasible solution. It is either provided by the user or determined during a first phase.
- Branching is performed on original variables, therefore not affecting the subproblem.
- The availability of the original variables in the column generation solver is the key to being able to use this solver in further hybrids.

Trucking Experiments

Finite domain model versus linearised branch-and-bound versus Column Generation.

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

Identical Subproblems

- Solving problems with identical subproblems by the pure Dantzig-Wolfe approach can be inefficient.
- We therefore aggregate identical subproblems:
  - The set K of subproblems is partitioned into sets K^s by grouping identical subproblems.
  - 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 λ_p^s are integer variables satisfying $0 \leq \lambda_p^s \leq |K^s|$ and $\sum_{p \in P^s} \lambda_p^s = |K^s|$.

Identical Subproblems

- The MP becomes the *Aggregated Master Problem*:

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

Automatic Disaggregation

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

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

- This usually leads to highly fractional values for the original variables, even if the λ_p^s are integer.
- We therefore decompose the λ_p^s values into λ_p^k values preserving integrality as much as possible, and then we use the mapping for the non-aggregated case.

Automatic Disaggregation

- In order to allow branching on the original variables the problem is disaggregated as required by the branching.
- It is possible to post any kind of linear constraint on the original variables without affecting the subproblem.
- Each aggregated subproblem appearing in these constraints is automatically disaggregated.
- If a constraint is posted involving an original variable belonging to a specific subproblem, this subproblem becomes different to the others and is disaggregated.

The Cutting Stock Problem

CuttingStock.zinc

```
int: K; int: N; int: L;
type Pieces = 1..K ::colgen_symmetric;
type Items  = 1..N;
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 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") );

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

The Cutting Stock Problem

A CADMIUM transformation creates an aggregated version:

CuttingStockAgg.zinc (changes)

```
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 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(0);

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

Cutting Stock

Experimental results with a maximum run-time of 5 min.

Class	Items	No Aggregation				Aggregation			
		Opt %	Feas %	Obj	Time [s]	Opt %	Feas %	Obj	Time[s]
Class1	10	30	70	12.70	210.40	30	70	12.60	209.95
Class2	10	70	10	118.75	100.89	90	10	112.90	59.36
Class3	20	30	0	23.33	242.52	20	80	24.50	250.05
Class4	20	0	0	n.a.	298.63	10	30	222.50	268.17
Class5	10	100	0	49.50	6.07	100	0	49.50	0.32
Class6	10	80	10	518.56	68.39	100	0	494.90	21.84
Class7	20	70	20	90.22	105.18	90	10	90	50.00
Class8	20	60	0	947.83	184.24	90	10	893.50	30.51
Class9	10	100	0	64	2.04	100	0	64	1.79
Class10	10	80	10	657.67	70.08	90	10	639.70	39.27
Class11	20	70	10	117.75	95.10	80	20	115.50	60.15
Class12	20	70	10	1182.25	154.79	80	20	1146.90	50.06
Average		63.33	11.67	330.74	128.19	73.33	21.67	327.46	86.79

Increase from 75% to 95% of solved instances.

Specialised Branching Rules

- Specialised branching rules for specific problem types were developed to overcome symmetry.
- 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 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.

Two-Dimensional Bin Packing

- We implemented a simple, well-known rule for the two-dimensional bin packing problem.
- The solution space is divided by branching on whether two different items are in the same bin.

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;
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"));

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

Two-Dimensional Bin Packing

Experimental results for two-dimensional bin packing with a maximum run-time of 5 min.

Class	Std. Branching				Sp. Branching			
	Opt %	Feas %	Obj	Time [s]	Opt %	Feas %	Obj	Time[s]
Class1	68	22	19.49	109.90	90	8	39.90	53.54
Class2	26	0	1.31	223.24	30	2	64.19	203.08
Class3	70	10	13.05	116.37	84	8	13.85	82.90
Class4	26	0	1.31	228.76	26	0	1.31	228.74
Class5	84	6	17.40	69.65	90	2	17.61	53.13
Class6	24	0	1.08	228.03	24	0	1.08	227.97
Class7	76	16	16.30	80.52	88	10	16.78	57.52
Class8	78	10	15.73	89.48	84	6	15.98	77.04
Class9	96	4	42.62	13.94	100	0	42.60	2.17
Class10	48	4	7.46	155.95	52	0	7.46	149.39
Average	59.6	7.2	17.95	131.58	66.8	3.6	23.65	113.55

Increase from 66.8% to 70.4% of solved instances.

Conclusions and Outlook

- G12 provides a framework for easy experimentations with Column Generation and Branch and Price.
- Different strategies for avoiding symmetries.
- Combination of different solvers allowing easy algorithm hybridisation.

- Address ipmlementation challenges such as column management and more sophisitcated branching rules.
- Specifying specialised branching rule in Zinc.
- Other problem decomposition methods such as Lagrangian or Benders decomposition.