Experiments with a Generic Dantzig-Wolfe Decomposition for Integer Programs

Gerald Gamrath[1] and Marco E. Lübbecke[2]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany, gamrath@zib.de
[2] Technische Universität Darmstadt, Fachbereich Mathematik, Dolivostr. 15, 64293 Darmstadt, Germany, luebbecke@mathematik.tu-darmstadt.de

Abstract We report on experiments with turning the branch-price-and-cut *framework* SCIP into a generic branch-price-and-cut *solver*. That is, given a mixed integer program (MIP), our code performs a Dantzig-Wolfe decomposition according to the user's specification, and solves the resulting re-formulation via branch-and-price. We take care of the column generation subproblems which are solved as MIPs themselves, branch and cut on the original variables (when this is appropriate), aggregate identical subproblems, etc. The charm of building on a well-maintained framework lies in avoiding to re-implement state-of-the-art MIP solving features like pseudo-cost branching, preprocessing, domain propagation, primal heuristics, cutting plane separation etc.

1 Situation

Over the last 25 years, branch-and-price algorithms developed into a very powerful tool to optimally solve huge and extremely difficult combinatorial optimization problems. Their success relies on exploiting problem structures in an integer program (via a decomposition or re-formulation) to which standard branch-and-cut algorithms are essentially "blind." While both, commercial and open-source solvers feature very effective generic implementations of branch-and-cut, almost every application of branch-and-price is *ad hoc*, that is, problem specific. Even though the situation improved considerably due to the availability of open-source branch-price-and-cut frameworks, implementations are often started from scratch or from previous projects. In addition, even though all concepts are reasonably easy to understand, experience and expert knowledge is still indispensable to get most out of the approach. In order to easily test new ideas, while using the current state-of-the-art, it would be much more satisfactory to have a generic implementation. This—ideally—performs a decomposition if this is likely to be promising, and takes care of branch-and-price (not to forget: -and-cut), without the user's notice and interaction—just as it is the case for generic branch-and-cut today. A future solver could terminate with the message

```
Integer optimal solution found (1209.71 sec., 2 threads)
Mixed integer rounding cuts applied: 65
Dantzig-Wolfe decomposition performed (3 subproblems)
```

While one may be sceptical about such a complete automatism (it requires at least detecting decomposable structures, and deciding how to best exploit them), a publicly available generic implementation which requires only a little user interaction is rather a matter of months than years from now. Our work is a contribution to this aim.

Related Work. There are several frameworks which support the implementation of branch-and-price algorithms like `ABACUS` [7], `BCP` [13], and `MINTO` [9], to name only a few. We restrict attention to (non-commercial) codes which perform a Dantzig-Wolfe decomposition of a general (mixed) integer program, and handle the resulting column generation subproblems in a generic way. François Vanderbeck has been developing important features [16,18,19,20] for his own implementation called `BaPCod` [17] which is a "prototype code that solves mixed integer programs (MIPs) by application of a Dantzig-Wolfe reformulation technique." Also the COIN-OR initiative (`www.coin-or.org`) hosts a generic decomposition code, called `DIP` [12] (formerly known as `DECOMP`), which is a "framework for implementing a variety of decomposition-based branch-and-bound algorithms for solving mixed integer linear programs" as described in [11]. The constraint programming `G12` project develops "user-controlled mappings from a high-level model to different solving methods," one of which is branch-and-price [10]. As of this writing, among these projects (including ours), only `DIP` is open to the public (as trunk development, there is no release yet).

Our Approach. We witness a development towards generic, re-usable implementations, however, in a sense, *again* started from scratch each time, at least in terms of standard MIP techniques like preprocessing and tree management etc. This is why, in this paper, we follow the different approach in complementing an *existing*, well-accepted, and well-maintained MIP solver, namely `SCIP` [1]. The rationale behind this is, of course, to have the full range of MIP tools available in one solver one future day, including a generic and automatic decomposition; so we found it reasonable to start from the state-of-the-art in non-commercial MIP solving [8]. The implementation is in such a way that it benefits from improvements of the solver itself, e.g., when better branching or node selection rules become available, preprocessing or propagation are getting more effective.

2 Decomposition of Integer Programs

We wish to solve an MIP, which is called the *original* (or *compact*) problem,

$$\min\{c^t x \mid Ax \geq b,\ Dx \geq d,\ x \in \mathbb{Z}^n_+ \times \mathbb{Q}^q_+\}\ . \tag{OP}$$

It exposes a *structure* in the sense that $X = \{x \in \mathbb{Z}^n_+ \times \mathbb{Q}^q_+ \mid Dx \geq d\}$ is a mixed integer set, optimization over which is considerably easier (computationally)

than solving (OP) itself. For many problems, D can be brought into a (bordered) block-diagonal form, so that (OP) can be written as

$$\min\{\sum_k c_k^t x^k \mid \sum_k A^k x^k \geq b,\ D^k x^k \geq d_k\ \forall k,\ x^k \in \mathbb{Z}_+^{n_k} \times \mathbb{Q}_+^{q_k}\ \forall k\}\ . \quad (\text{OP}k)$$

In this case $X = X_1 \times \cdots \times X_K$ (possibly permuting variables), that is, X *decomposes* into $X_k = \{x^k \in \mathbb{Z}_+^{n_k} \times \mathbb{Q}_+^{q_k} \mid D^k x \geq d_k\}$, $k = 1, \ldots, K$, with all matrices and vectors of compatible dimensions and $\sum_k n_k = n$, $\sum_k q_k = q$. We discuss two ways of exploiting this structure when solving (OP). A very thorough exposition of advantages and disadvantages, possibilities, extensions, examples, and much more context can be found in the most recent survey [22].

Convexification. By the Minkowski-Weyl theorem we express each $x^k \in \text{conv}(X_k)$ as a convex combination of extreme points P_k of $\text{conv}(X_k)$ plus a non-negative combination of extreme rays R_k of $\text{conv}(X_k)$, with P_k and R_k finite. For ease of presentation, we assume X_k bounded, i.e., $R_k = \emptyset$. In analogy to a Dantzig-Wolfe decomposition of linear programs, we introduce a variable λ_p^k for each $p \in P_k$ and require $\sum_{p \in P_k} \lambda_p^k = 1$ (*convexity constraints*). Substituting $x^k = \sum_{p \in P_k} p\lambda_p^k$, we obtain the *extended* formulation

$$\begin{aligned}\min\{\sum_k \sum_{p \in P_k} c_p^k \lambda_p^k \mid \sum_k \sum_{p \in P_k} a_p^k \lambda_p^k \geq b,\ \sum_{p \in P_k} \lambda_p^k = 1,\ x^k = \sum_{p \in P_k} p\lambda_p^k\ \forall k,\\ x^k \in \mathbb{Z}_+^{n_k} \times \mathbb{Q}_+^{q_k}\ \forall k,\ \lambda^k \in \mathbb{Q}_+^{|P_k|}\ \forall k\}\end{aligned} \quad (\text{EPC})$$

where $c_p^k = c_k p$ and $a_p^k = A_k p$. Integrality is required on the original variables.

Discretization. For pure integer programs, i.e., $q_k = 0$ for all k, one can implicitly express x^k as an *integer* convex combination of the integer points in X_k, i.e., $x^k = \sum_{p \in X_k} p\lambda_p^k,\ \lambda_p^k \in \{0,1\}\ \forall k$ [16]. Unifying the notation with the convexification approach, we denote the set X_k of points by P_k and obtain

$$\min\{\sum_k \sum_{p \in P_k} c_p^k \lambda_p^k \mid \sum_k \sum_{p \in P_k} a_p^k \lambda_p^k \geq b,\ \sum_{p \in P_k} \lambda_p^k = 1,\ \lambda^k \in \mathbb{Z}_+^{|P_k|}\ \forall k\}\ . \quad (\text{EPD})$$

This can be generalized to MIPs, when continuous variables are convexified [21]. Often, some or all X_k are identical, e.g., for bin packing, vertex coloring, or some vehicle routing problems. This introduces a symmetry which is avoided by aggregating (summing up) the λ_p^k variables. We choose a representative $P := P_1$, substitute $\lambda_p := \sum_k \lambda_p^k$, and add up the convexity constraints. This leads to the *aggregated extended formulation*

$$\min\{\sum_{p \in P} c_p \lambda_p \mid \sum_{p \in P} a_p \lambda_p \geq b,\ \sum_{p \in P} \lambda_p = K,\ \lambda \in \mathbb{Z}_+^{|P|}\}\ . \quad (\text{EPDa})$$

Column Generation and Branch-and-Price. For the LP relaxation of the extended problem, we drop the integrality constraints, and also omit the original variables in the convexification approach. We obtain the *master problem*

$$\min\{\sum_k \sum_{p\in P_k} c_p^k \lambda_p^k \mid \sum_k \sum_{p\in P_k} a_p^k \lambda_p^k \geq b,\ \sum_{p\in P_k} \lambda_p^k = 1,\ \lambda^k \in \mathbb{Q}_+^{|P_k|}\ \forall k\}\ . \qquad \text{(MP)}$$

Since (MP) typically has an exponential number of variables, it is solved via column generation. That is, we work with a *restricted master problem (RMP)* that contains only a subset of the variables. In each node of the branch-and-bound tree, the RMP is solved to optimality, and variables with negative reduced cost are added. One iterates until no more variables are found. As the reduced cost of a variable λ_p^k is given by $\bar{c}_p^k = c_k^t p - (\pi^t A^k p + \gamma_k)$ with $(\pi^t, \gamma^t)^t$ being the optimal dual solution to the RMP, we solve, for each block $k \in [K]$, the *pricing problem* $\bar{c}_k^\star = \min\left\{\left(c_k^t - \pi^t A^k\right) x - \gamma_k \mid x \in X_k\right\}$. The LP relaxation can be strengthened by additional valid inequalities (in different ways). If the solution is still fractional, branching takes places, for convexification typically on the original variables, but see e.g., [20] for a different generic rule which does not interfere with the pricing problem and avoids symmetry.

3 Some Details on the Implementation in SCIP

Our implementation GCG (generic column generation) extends SCIP [1] which was well received in the computational mathematical programming community. While the flexible plugin-based architecture enables the user to easily implement column generation in every node of the search tree, it neither provides methods for decomposition nor does it work with original and extended problem formulations simultaneously. Our work aims at complementing SCIP in this respect, turning the branch-price-and-cut *framework* into a branch-price-and-cut *solver*.

3.1 Overview: Synchronizing two Trees

We maintain two SCIP *instances*, one for the original, one for the extended problem (called original and extended instance, respectively). The original instance is the primary one which coordinates the solving process, the extended instance is controlled by a relaxation handler that is included into the original instance. At the moment, information about the structure of the problem has to be provided by an additional input file, that defines the relation between variables and blocks and may explicitly force constraints as linking constraints, i.e., constraints that will be transferred to the extended (master) problem.

After the original instance is presolved, the relaxator performes the Dantzig-Wolfe decomposion and initializes the extended SCIP instance as well as the SCIP instances representing the pricing problems. The extended instance initially contains no variables. Original variables that are labeled to be part of a block, and constraints containing variables of just one block are copied into the

corresponding pricing problem unless explicitly forced otherwise. All remaining constraints are transferred to the extended problem.

During the solving process, the extended instance builds the branch-and-bound tree in the same way as the original instance. There is a bijection between nodes of the original instance and nodes of the extended instance; two corresponding nodes are solved at the same time. When solving a node of the original instance, the node lower bound is not computed by solving the node's LP relaxation, but the special relaxator is used for this purpose which instructs the extended SCIP instance to solve the next node. A special node selector in the extended instance chooses as the next node to be processed the node corresponding to the current node in the original instance. Branching restrictions are imposed by a branching rule included in the original instance, so they have to be transferred to the node of the extended instance when it is activated. The solving process of the node starts with domain propagation, i.e., tightening the domains of the variables for the local problem, a concept that is also used for the enforcement of branching decisions. The LP relaxation of the problem corresponding to the node—the master problem—is then solved by column generation.

After the master problem is solved, bounding is performed and branching is performed if needed, creating two children without further problem restrictions. The solving process of the extended problem is then halted and the relaxator in the original instance transfers the local dual bound and the master problem's current solution as well as new primal solutions to the original instance. The current node in the original instance is pruned if and only if the corresponding node in the extended instance was pruned, since both nodes have the same dual bound and both instances have the same primal bound—each solution of one instance corresponds to a solution of the other instance with the same objective function value. However, it is possible that the master solution is fractional but leads to an integral solution to the original problem. In this case, the current subproblem is solved to optimality, otherwise, branching is performed. Two children are created and branching restrictions are imposed in the original instance, that will be transferred to the corresponding nodes in the extended instance on activation. After the branching, the original instance selects a next node and the process is iterated. Fig. 1 shows GCG's solving process.

We decided to work with both formulations simultaneously rather than transforming the original problem into an extended problem and solving this problem with a branch-price-and-cut algorithm, since it fits better into the SCIP framework and makes better use of the functionalities already provided. The original problem can be read in a variety of formats by the original instance using SCIP's default file reader plugins. If we do not read a file defining the structure of the problem, the problem is solved by SCIP with a branch-and-cut algorithm. Otherwise, the special relaxator is activated, creates the extended SCIP instance, performs the Dantzig-Wolfe decomposition, and substitutes the LP relaxation in the branch-and-bound process. Both problems are solved in parallel, so that techniques that speed up the solving process, like presolving, domain propagation, and heuristics, can be used in both instances. For details see [6].

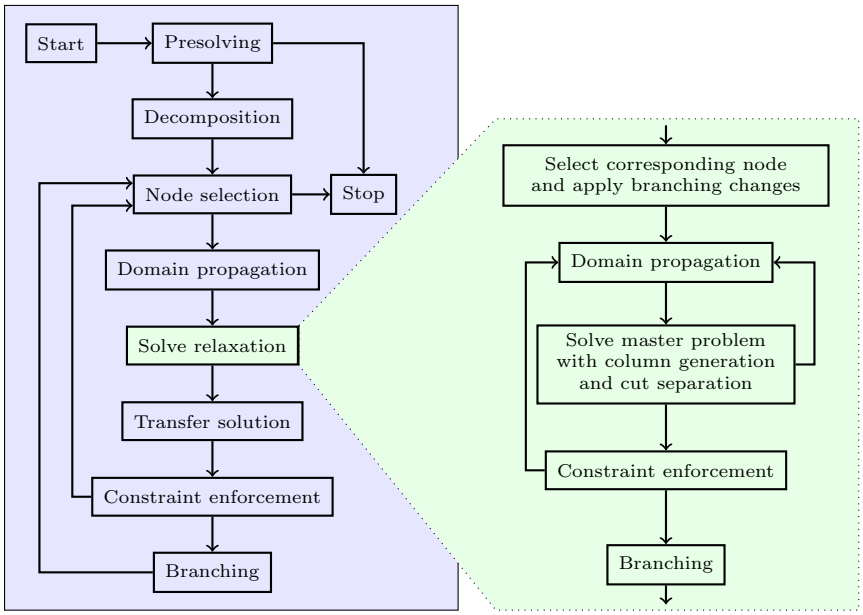

Figure 1. Solving process of GCG

3.2 Pricing

We structure the pricing implementation in *variable pricer* and a set of *pricing solvers*. The former coordinates the pricing process, while the latter are called by the variable pricer to solve a specific pricing problem. A variable pricer plugin is added to the extended instance. These plugins have two essential callbacks that are called during the pricing process, one for *Farkas pricing*, which is called by SCIP whenever the RMP is infeasible, the other for the reduced cost pricing, which is called in case the RMP is feasible.

We introduced the concept of pricing solvers, which are used by the pricer in a black box fashion: Whenever a specific pricing problem should be solved, it is given to the set of solvers, solved by one of the solvers, and a set of solutions is returned. We chose this concept, which is similar to the way the LP solver is handled in SCIP, in order to provide a possiblity to add further problem specific solvers as external plugins without the need to modify the variable pricer.

We compute the intermediate Lagrangean dual bounds every time all pricing problems were solved to optimality in a pricing round and update the dual bound of the current node each time this leads to an improvement. We make use of *early termination*, i.e., if all solutions have integral values—this is detected in the presolving process of the original problem—we abort the pricing process at a node whenever $\lceil LB \rceil \geq z_{RMP}$ for the current local dual bound LB and the optimal objective value z_{RMP} of the RMP in this pricing iteration.

When using the discretization approach, we identify identical subproblems and automatically aggregate them. That is, if $X_i = X_j$ for all $i, j \in \{1, \ldots, k\}$, we define aggregate variables $\lambda_p = \sum_k \lambda_p^k$ in the master problem.

3.3 Branching Rules

We provide two branching rules, one that branches on the variables of the original problem and Ryan and Foster's branching scheme [14] for problems with a set partitioning master problem. When branching on original variables, the master solution is transferred into a solution of the original problem, an integer variable x_i^k with fractional value v is identified and the domain of the variable is split by adding constraints $x_i^k \geq \lceil v \rceil$ and $x_i^k \leq \lfloor v \rfloor$, respectively, to the two child nodes. These constraints can be enforced in the pricing problems as well as in the extended problem. The latter leads to an additional constraint in the master problem, its dual variable is respected in the objective function of the pricing problem like it is done for the other master constraints. Enforcing the branching decisions in the pricing problems can lead to better dual bounds at the expense that all variables in the master problem have to be checked for their feasibility w. r. t. the current pricing problems. This is done via domain propagation in the extended instance; variables that are not compatible with branching decisions are locally fixed to zero. It is well-known that the choice of the variable to branch on has a big impact on the performance of the branch-and-bound process [2]. Hence, apart from most infeasibility branching, we provide the possibility to make use of pseudocosts of the variables in the original problem. This leads to a considerable decrease of computational time for the class of capacitated p-median problems (Sect. 4.1). The Ryan and Foster branching scheme is used for problems with a set partitioning master structure and for problems with identical blocks.

3.4 Presolve and Propagation

We perform standard `SCIP` presolving on the original problem. Furthermore, at each node, we perform domain propagation in the original instance as well as the extended instance. In the extended instance, we use it primarily to remove variables from the problem, that are not valid for the current pricing problem. In the original instance, standard domain propagation methods are used that lead to domain reductions especially when branching on the original variables. These reductions can then be imposed on the variables of the pricing problems, too. Variables that fulfill these bounds are called *proper* [21].

3.5 Cutting Plane Separators

A by now "standard" way to strengthen the dual bound it to derive valid inequalities from the original variables. An original fractional solution can be separated by `SCIP`'s default cut separation plugins. Note that $x^k = \sum_{p \in P_k} p\lambda_p^k$ is not a basic solution in (OPk), thus we cannot use *Gomory mixed integer cuts* or *strong*

Chvátal-Gomory cuts. Nevertheless, we can use all kinds of cutting planes that do not need any further information besides the problem and the current solution in their separation routine. This applies, for example, to *knapsack cover cuts*, *mixed integer rounding cuts*, and *flow cover cuts*. An alternative is to derive cuts from the extended formulation. This may lead to stronger cuts but has the drawback that the dual variable of a cut has to be respected in the pricing problems, which typically results in additional variables and constraints added to the pricing problems, see [5] for a very recent unified view.

3.6 Primal Heuristics

The default primal heuristics of `SCIP` (including sophisticated ones) are applied to the extended instance. Simple rounding heuristics are performed in each iteration of the column generation process which often find feasible primal solutions. Currently, most heuristics make use of the LP relaxation, so we cannot run them on the original instance. This is to be changed in the future.

3.7 Customization and Extendibility

It is to be expected that a tailor-made approach will outperform a generic one, so we keep the possibility to extend and customize the framework, in addition to adding pricing solvers. `SCIP`'s interface for branching rules is extended in order to give the possibility to define branching rules operating on both the original as well as the extended formulation. This includes ways to enforce branching decisions in the pricing problems and to store pseudocosts for branching on constraints. Finally, problem specific plugins for presolving, node selection, domain propagation, separation and primal heuristics can be added to either one of the two `SCIP` instances as usual.

4 Computational Study

We tested our solver `GCG` 0.7 on MIPs which expose various different structures using `SCIP` 1.2.0.5 with `CPLEX` 12.1 as embedded LP solver. The computations presented in Section 4.1 and 4.2 were performed on a 2.66 Ghz Core2 Quad with 4MB Cache and 4GB RAM, those of Section 4.3 on a 2.83 GHz Core2 Quad with 6MB cache and 16GB RAM. We compute averages using the shifted geometric mean, i.e., for non-negative numbers $a_1, \ldots, a_k \in \mathbb{R}_+$, e.g., the number of nodes, the solving time, or the final gap of the individual instances of a test set, and a shift $s \in \mathbb{R}_+$, the average is defined by

$$\gamma_s(a_1, \ldots, a_k) = \left(\prod_{i=1}^{k} (a_i + s) \right)^{\frac{1}{k}} - s.$$

We use a shift of 10 for the runtime and the number of branch-and-bound nodes and 100 for the final gap in percent. The average value of multiple test sets is computed in the same way, using the shifted geometric means of the individual test sets.

	test set	SCIP	GCG	no pseudocost	knapsack
nodes	CPMP50S	896.2	44.9	82.5	42.1
	CPMP100S	14234.2	587.1	1962.6	491.6
	CPMP150S	15128.7	847.6	2211.7	1228.6
	CPMP200S	26263.9	1753.3	5577.7	2338.0
	sh. geom. mean	8454.9	461.8	1216.6	515.1
time (outs)	CPMP50S	14.5 (0)	12.8 (0)	20.7 (0)	1.8 (0)
	CPMP100S	234.8 (3)	184.7 (1)	469.5 (6)	37.8 (0)
	CPMP150S	714.5 (9)	493.5 (5)	920.3 (10)	253.8 (1)
	CPMP200S	1950.7 (7)	1243.9 (3)	2978.0 (10)	519.3 (0)
	sh. geom. mean	294.0 (19)	220.1 (9)	439.7 (26)	84.3 (1)

Table 1. Comparison of GCG and SCIP for the capacitated p-median test sets. We list the shifted geometric mean of the number of branch-and-bound nodes (top), and the runtime (bottom) for SCIP (first column) and GCG (second column) with default settings. The next columns illustrate the performance effect of disabling the pseudocost branching rule and using the most fractional rule instead (third column) and of using a specialized knapsack solver to solve the pricing problems (last column). Following the runtime, in brackets, we list the absolute number of timeouts.

4.1 Different Subproblems: The Capacitated p-Median Problem

In the *capacitated p-median problem* we are given a set N of nodes, each with a demand $q_n \in \mathbb{Z}_+$, $n \in N$. In each node $n \in N$, a facility with capacity C can be opened; p of which have to opened in total. The distance between a node $n \in N$ and a facility placed at node $m \in N$ is given as $d_{n,m} \in \mathbb{Z}$. Nodes are assigned to opened facilities so that the total sum of connection distances is minimized and the capacity constraints are respected. To solve large instances by branch-and-price, so far an *ad hoc* implementation was necessary [4].

The problem can be decomposed by defining one block for each possible facility location which contains the capacity constraint corresponding to this location. The blocks are not identical since the objective function coefficients of the variables depend on the location represented by this block. Therefore, we branch on the original variables. We used a subset of the instances used in [4] and defined four test sets CPMPNS, each of which contains instances with N nodes, $N \in \{50, 100, 150, 200\}$. In order to reduce the computational effort, we missed out every second instance in the test sets with up to 150 nodes and selected only 12 instances for test set CPMP200S, three for each number of facilities.

Tab. 1 shows that the generic branch-and-price approach performs better than plain SCIP. It particularly pays off to have a state-of-the-art branching rule at hand: using most fractional branching instead of pseudocost branching doubles the shifted geometric mean of the solution time. Furthermore, by using a dynamic programming knapsack solver to solve the pricing problems and customizing the code in this way, we are able to decrease the shifted geometric mean of the solving time by more than 60% compared to GCG with default settings. More detailed computational experiments are reported in [6].

number of items	nodes total	nodes sh. geom. mean	time total	time sh. geom. mean
50	713	3.3	54.7	0.3
100	1617	6.7	254.4	1.4
200	5229	17.6	2186.8	11.7

Table 2. Computational results for the 180 instances of each size in the bin packing test set.

4.2 Identical Subproblems: Bin Packing

Bin packing instances have identical blocks and a set partitioning master problem, so the variables were aggregated and Ryan and Foster's branching scheme was used. It is well-known that the Dantzig-Wolfe decomposition of the bin packing problem leads to strong dual bounds, so we were able to solve all 540 instances (all 180 instances with 50, 100, and 200 items, respectively, of data set 1 of [15]) in less than 90 minutes altogether, cf. Tab. 2. For each number of items, `GCG` solves the whole test set faster than `SCIP` solves the first instance of the set.

4.3 No Block Structure: A Resource Allocation Problem

The following generalized knapsack problem [3] does not have a block structure (but staircase structure). Given a number of periods $n \in N$ and items $i \in I$, each item has a profit p_i, a weight w_i, and a starting and ending period. In each period, the knapsack has capacity C and items consume capacity only during their *life time*. The problem can be modelled in the following way:

$$\max\{\sum_{j\in I} p_i x_i \mid \sum_{i\in I(n)} w_i x_i \leq C \ \forall n \in N,\ x_i \in \{0,1\}\ \forall i \in I\}\ , \tag{RAP}$$

where $I(n)$ is the set of items that are alive in period $n \in N$. The matrix can be transformed into block structure by splitting the capacity constraints into groups of size M [3]. For each variable that appears in more than one group, we create a copy of this variable for each group and link the values of these copies to each other by additional constraints. These additional constraints will be part of the extended formulation, the M capacity constraints corresponding to a block are transferred to this block's pricing problem.

We performed computational experiments (see Tab. 3) for the instances described in [3]. We used `SCIP` 1.2.0.5 for solving formulation RAP explicitly, and `GCG` 0.7 to solve the reformulation of the problem grouping 32 and 64 constraints to form one block, respectively. The same grouping was used in [3]. `SCIP` was able to solve five instances within the timelimit of one hour, the remaining 65 instances remained unsolved with a final gap between 0.1 and 3.0 percent. For both numbers of constraints grouped per block, `GCG` was able to solve 56 instances, the final gap of the remaining instances was typically lower than 0.3

percent. For both sizes of blocks, GCG was about four times faster than SCIP in the shifted geometric mean.

The relaxation given by the master problem is tighter the more constraints are assigned to a block, so with 64 constraints per block, we need less nodes to solve the problems. The shifted geometric mean of the number of nodes accounts 11.5 for the former variant, compared to 21 nodes when assigning 32 constraints to each block. In return, more time is needed to solve the master problem, but this pays off for this test set since the total time is reduced by 8%. The average gap is higher when grouping 64 constraints, however, this is caused by one single instance for which the master problem at the root node could not be solved within the time limit of one hour so that just a trivial dual bound is obtained, leading to a gap of more than 70 percent.

5 Summary and Discussion

We report first computational experiments with a basic generic implementation of a branch-price-and-cut algorithm within the non-commercial framework SCIP. Given an MIP and information about which rows belong to which subproblem (or the master problem), either a convexification or discretization style decomposition is performed. For structured problems, the approach is very promising.

The modular design of SCIP allowed us to include the described functionality in the form of *plugins*. A true integration would require a few extensions, some of which have been incorporated into SCIP during this project already, but some are still missing. Examples include per-row dual variable stabilization, column pool, primal heuristics on original variables, LP basis of original variables for cutting planes like Gomory cuts, etc. It is planned that our implementation becomes part of a future release of SCIP. We hope that this enables researchers to play with and quickly test ideas in decomposing mixed integer programs.

It remains to be demonstrated that there really is a significant share of problems on which decomposition methods are more effective than (or a reasonable complement to) standard branch-and-cut, even when one does not know about a possibly contained structure. This requires detecting whether it may pay to decompose any given MIP, and if so, how this should be done. This is, of course, a much more challenging question which is the subject of our current research.

Acknowledgment We thank Alberto Ceselli and Enrico Malaguti for providing us with the p-median and RAP instances from Sections 4.1 and 4.3, respectively.

References

1. T. Achterberg, *SCIP: Solving constraint integer programs*, Math. Programming Computation, 1 (2009), pp. 1–41.
2. T. Achterberg, T. Koch, and A. Martin, *Branching rules revisited*, Operations Research Letters, 33 (2005), pp. 42–54.

3. A. Caprara, F. Furini, and E. Malaguti, *Exact algorithms for the temporal knapsack problem*, Technical report OR-10-7, DEIS, University of Bologna, 2010.
4. A. Ceselli and G. Righini, *A branch-and-price algorithm for the capacitated p-median problem*, Networks, 45 (2005), pp. 125–142.
5. G. Desaulniers, J. Desrosiers, and S. Spoorendonk, *Cutting planes for branch-and-price algorithms*, Les Cahiers du GERAD G-2009-52, HEC Montréal, 2009.
6. G. Gamrath, *Generic branch-cut-and-price*, Master's thesis, Institut für Mathematik, Technische Universität Berlin, 2010.
7. M. Jünger and S. Thienel, *The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization*, Softw. Pract. Exper., 30 (2000), pp. 1325–1352.
8. H. Mittelmann, *Benchmarks for optimization software*. `http://plato.asu.edu/bench.html`, 2010.
9. G. Nemhauser, M. Savelsbergh, and G. Sigismondi, *MINTO, a Mixed INTeger Optimizer*, Oper. Res. Lett., 15 (1994), pp. 47–58.
10. J. Puchinger, P. Stuckey, M. Wallace, and S. Brand, *Dantzig-Wolfe decomposition and branch-and-price solving in G12*, Constraints, (2010). To appear.
11. T. Ralphs and M. Galati, *Decomposition and dynamic cut generation in integer linear programming*, Math. Programming, 106 (2006), pp. 261–285.
12. ———, *DIP – decomposition for integer programming*. `https://projects.coin-or.org/Dip`, 2009.
13. T. Ralphs and L. Ladányi, *COIN/BCP User's Manual*, 2001. `http://www.coin-or.org/Presentations/bcp-man.pdf`.
14. D. Ryan and B.A.Foster, *An integer programming approach to scheduling*, in Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling, A. Wren, ed., North Holland, Amsterdam, 1981, pp. 269–280.
15. A. Scholl and R. Klein, *Bin packing instances: Data set 1*. `http://www.wiwi.uni-jena.de/Entscheidung/binpp/`.
16. F. Vanderbeck, *On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm*, Oper. Res., 48 (2000), pp. 111–128.
17. ———, *BaPCod – a generic branch-and-price code*. `https://wiki.bordeaux.inria.fr/realopt/pmwiki.php/Project/BaPCod`, 2005.
18. ———, *Implementing mixed integer column generation*, in Column Generation, G. Desaulniers, J. Desrosiers, and M. Solomon, eds., Springer, 2005, pp. 331–358.
19. ———, *A generic view of Dantzig-Wolfe decomposition in mixed integer programming*, Oper. Res. Lett., 34 (2006), pp. 296–306.
20. ———, *Branching in branch-and-price: A generic scheme*, Math. Programming, (2010). To appear.
21. F. Vanderbeck and M. Savelsbergh, *A generic view of Dantzig-Wolfe decomposition in mixed integer programming*, Oper. Res. Lett., 34 (2006), pp. 296–306.
22. F. Vanderbeck and L. Wolsey, *Reformulation and decomposition of integer programs*, in 50 Years of Integer Programming 1958–2008, M. Jünger, T. Liebling, D. Naddef, G. Nemhauser, W. Pulleyblank, G. Reinelt, G. Rinaldi, and L. Wolsey, eds., Springer, Berlin, 2010.

	SCIP			GCG (32)			GCG (64)		
instance	gap	nodes	time	gap	nodes	time	gap	nodes	time
new1_1	1.3	>757186	>3600.0	0.0	29	201.2	0.0	3	111.4
new1_2	1.7	>589833	>3600.0	0.0	119	979.6	0.0	13	350.5
new1_3	1.6	>408635	>3600.0	0.0	23	359.7	0.0	9	529.5
new1_4	1.7	>296254	>3600.0	0.0	187	1888.9	0.0	7	473.3
new1_5	2.0	>262849	>3600.0	0.0	41	828.6	0.0	11	562.3
new1_6	1.4	>194120	>3600.0	0.0	17	468.5	0.0	39	1143.4
new1_7	1.5	>163145	>3600.0	0.0	27	460.6	0.0	1	478.7
new1_8	1.7	>161800	>3600.0	0.0	13	583.6	0.0	3	350.6
new1_9	1.8	>108307	>3600.0	0.0	87	1687.5	0.0	2	453.4
new1_10	2.2	>84190	>3600.0	0.0	35	1085.1	0.0	7	896.1
new2_1	2.0	>750775	>3600.0	0.0	7	109.5	0.0	3	140.0
new2_2	1.4	>580746	>3600.0	0.0	136	898.0	0.0	44	744.7
new2_3	1.1	>415357	>3600.0	0.0	23	299.5	0.0	2	135.9
new2_4	1.2	>428950	>3600.0	0.0	1	118.9	0.0	1	367.1
new2_5	1.9	>207493	>3600.0	0.0	161	1622.6	0.1	>154	>3600.0
new2_6	2.0	>198288	>3600.0	0.0	11	425.6	0.0	4	326.9
new2_7	1.6	>188555	>3600.0	0.0	36	642.2	0.0	13	740.3
new2_8	1.7	>142970	>3600.0	0.0	13	399.6	0.0	14	782.2
new2_9	1.8	>75900	>3600.0	0.0	31	702.4	0.0	1	293.5
new2_10	1.9	>131284	>3600.0	0.0	151	1489.0	0.0	41	1274.4
new3_1	1.3	>233708	>3600.0	0.0	31	1205.1	0.0	1	1205.8
new3_2	1.2	>89037	>3600.0	0.0	>119	>3600.0	0.0	>49	>3600.0
new3_3	1.3	>75770	>3600.0	0.0	13	1227.5	0.0	>7	>3600
new3_4	1.4	>84830	>3600.0	0.0	39	2596.2	0.0	9	1688.5
new3_5	1.7	>19150	>3600.0	0.0	33	1937.4	0.0	9	3039.5
new3_6	2.2	>8632	>3600.0	0.1	>55	>3600.0	0.0	>39	>3600.0
new3_7	1.8	>35455	>3600.0	0.0	>88	>3600.0	0.0	>62	>3600.0
new3_8	1.8	>30130	>3600.0	0.0	53	3535.1	0.0	5	3172.4
new3_9	2.4	>25500	>3600.0	0.1	>40	>3600.0	73.8	>1	>3600.0
new3_10	2.6	>864	>3600.0	0.0	>45	>3600.0	0.2	>27	>3600.0
new4_1	0.0	4907	16.1	0.0	1	26.1	0.0	1	15.6
new4_2	0.0	187	4.8	0.0	1	17.9	0.0	1	17.1
new4_3	0.0	132	6.0	0.0	1	24.6	0.0	1	28.4
new4_4	0.1	>1571331	>3600.0	0.0	1	73.0	0.0	1	44.5
new4_5	0.0	70140	342.6	0.0	13	86.4	0.0	13	106.3
new4_6	0.2	>792713	>3600.0	0.0	1	58.8	0.0	1	66.1
new4_7	0.0	670545	3406.9	0.0	1	57.0	0.0	1	54.1
new4_8	0.2	>535592	>3600.0	0.0	4	99.3	0.0	3	59.0
new4_9	0.2	>546140	>3600.0	0.0	3	92.6	0.0	3	93.6
new4_10	0.2	>507737	>3600.0	0.0	3	61.0	0.0	5	154.4
new5_1	0.7	>640409	>3600.0	0.0	23	254.4	0.0	1	130.1
new5_2	1.3	>317663	>3600.0	0.0	11	248.6	0.0	1	230.7
new5_3	0.9	>318430	>3600.0	0.0	3	177.1	0.0	3	282.4

cont'd next page

	SCIP			GCG (32)			GCG (64)		
instance	gap	nodes	time	gap	nodes	time	gap	nodes	time
new5_4	1.4	>247945	>3600.0	0.0	75	1445.6	0.0	3	265.3
new5_5	1.4	>120474	>3600.0	0.0	42	871.3	0.0	17	887.4
new5_6	1.3	>147990	>3600.0	0.0	161	3570.1	0.0	>129	>3600.0
new5_7	1.0	>135050	>3600.0	0.0	133	2008.2	0.0	13	730.5
new5_8	1.3	>114213	>3600.0	0.0	15	554.7	0.0	1	564.4
new5_9	1.2	>78548	>3600.0	0.0	>124	>3600.0	0.0	>87	>3600.0
new5_10	1.7	>22990	>3600.0	0.0	85	2109.2	0.0	25	1799.1
new6_1	1.0	>215663	>3600.0	0.0	25	692.2	0.0	7	484.2
new6_2	1.2	>159970	>3600.0	0.0	7	518.3	0.0	3	686.3
new6_3	1.0	>92896	>3600.0	0.0	27	975.0	0.0	11	559.0
new6_4	1.0	>93850	>3600.0	0.0	17	1091.2	0.0	1	628.5
new6_5	1.1	>69570	>3600.0	0.0	29	1546.7	0.0	21	1826.8
new6_6	1.8	>14540	>3600.0	0.0	13	1094.0	0.0	7	1714.0
new6_7	2.3	>10384	>3600.0	0.0	>51	>3600.0	0.0	3	1368.9
new6_8	1.6	>6209	>3600.0	0.1	>77	>3600.0	0.0	>38	>3600.0
new6_9	1.7	>38634	>3600.0	0.0	>37	>3600.0	0.0	3	2030.4
new6_10	3.0	>772	>3600.0	0.0	>55	>3600.0	0.0	>35	>3600.0
new7_1	1.4	>537230	>3600.0	0.0	58	614.3	0.0	7	276.9
new7_2	1.5	>373513	>3600.0	0.0	31	624.5	0.0	41	1263.9
new7_3	1.4	>230756	>3600.0	0.0	46	823.1	0.0	11	545.2
new7_4	1.8	>164797	>3600.0	0.0	25	665.2	0.0	>37	>3600.0
new7_5	1.3	>147376	>3600.0	0.0	9	486.2	0.0	9	791.8
new7_6	1.7	>158523	>3600.0	0.0	>186	>3600.0	0.0	34	3032.1
new7_7	1.7	>95711	>3600.0	0.0	>140	>3600.0	0.1	>74	>3600.0
new7_8	2.2	>96525	>3600.0	0.1	>148	>3600.0	0.0	19	3205.4
new7_9	1.7	>80942	>3600.0	0.0	31	1051.1	0.0	11	1800.4
new7_10	2.4	>78398	>3600.0	0.0	>107	>3600.0	0.1	>42	>3600.0
total	97.1	16259k	237776.4	0.4	3484.0	98169.6	74.3	1305.0	95403.1
timeouts			65/70			14/70			14/70
sh. geom. mean	1.4	97564.0	2772.4	0.0	32.2	727.4	0.8	11.5	670.7

Table 3. Computational results for the test set of RAP instances. We list the final gap, the number of branch-and-bound nodes and the runtime for SCIP, GCG with 32 constraints assigned to a block, and GCG with 64 constraints assigned to a block.