Implementing the branch-and-cut approach for a general purpose Benders' decomposition framework

Stephen J. Maher*

College of Engineering, Mathematics and Physical Sciences, University of Exeter, Exeter, United Kingdom

Abstract

Benders' decomposition is a popular mathematical and constraint programming algorithm that is widely applied to exploit problem structure arising from real-world applications. While useful for exploiting structure in mathematical and constraint programs, the use of Benders' decomposition typically requires significant implementation effort to achieve an effective solution algorithm. Traditionally, Benders' decomposition has been viewed as a problem specific algorithm, which has limited the development of general purpose algorithms and software solutions. This paper presents a general purpose Benders' decomposition algorithm that is capable of handling many classes of mathematical and constraint programs and provides extensive flexibility in the implementation and use of this algorithm. A branch-and-cut approach for Benders' decomposition has been implemented within the constraint integer programming solver SCIP using a plugin-based design to allow for a wide variety of extensions and customisations to the algorithm. The effectiveness of the Benders' decomposition algorithm and available enhancement techniques is assessed in a comprehensive computational study.

Key words: Benders' decomposition, branch-and-cut, mixed integer programming, constraint integer programming, optimisation software

1 Introduction

Benders' decomposition [7] (BD) is a classical technique that exploits structure in mathematical and constraint programs. Structured mathematical and constraint programs typically arise when modelling real world applications, such as airline planning [14,42,48], facility location [3,18], network design [12,53] and network interdiction problems [11, 46]. The use of decomposition techniques to exploit problem structure has led to many advancements in solution methods for real world optimisation problems.

Historically, there has been a belief that the application of decomposition techniques requires a problem specific algorithm—limiting the potential generalisation of the approaches. While problem

*s.j.maher@exeter.ac.uk

structure is most commonly known by the modeller, it is possible to view the fundamental features of decomposition algorithms in general terms. Separating structure detection from decomposition solution algorithms exposes a general algorithmic framework that can be applied to a large class of mathematical and constraint programs. This has lead to a growth in software solutions providing general frameworks for the application of decomposition techniques.

This paper presents a BD framework that has been developed for the constraint integer programming solver SCIP [25]. Central to the design of the framework is flexibility in the implementation and use of the BD algorithm. The core algorithmic structure can be employed with little input and implementation work by the researcher. Further, a plugin-based design provides many options for extension to meet the users needs, such as custom BD cut generation routines and subproblem solving methods. Supporting the core BD algorithm is a collection of classical and novel enhancement techniques. Through the use of the generic branch-cut-and-price solver Generic Column Generation (GCG) [8], structure detection and the automatic application of BD can be performed. The extension of GCG enables the use of the popular BD algorithm without any in-depth knowledge or expertise of decomposition methods. The BD framework is designed to be simple to use as well as provide sufficient flexibility to support the research community in extending our knowledge and understanding of the BD algorithm.

1.1 Benders' decomposition

Consider problems of the form

$$\min_{x,y}\{c^\top x + d^\top y \,|\, Ax \geq b, Bx + Dy \geq g, x \in \mathbb{Z}_+^p \times \mathbb{R}_+^{n-p}, y \in \mathbb{Z}_+^q \times \mathbb{R}_+^{m-q}\}. \tag{1}$$

Such problems are described as having a bordered block diagonal structure. The constraints $Ax \geq b$ correspond to the first stage and the constraints $Bx + Dy \geq g$ correspond to the second stage. Similarly, the first and second stage variables are given by x and y respectively. Both sets of first and second stage variables can be either discrete, continuous or a combination of the two. Note that structure can be exhibited by matrix D, such as block diagonal or bordered block diagonal, that can be exploited further when applying decomposition techniques.

The existence of the border given by x makes problem (1) suitable for the application of BD. The decomposition is applied by partitioning the constraints into a master problem and one or more subproblems—depending on the structure of D. Applying BD to problem (1) results in a subproblem comprising of the second stage constraints and the related variables, given by

$$z(\hat{x}) = \min_{y}\{d^\top y \,|\, Dy \geq g - B\hat{x}, y \in \mathbb{Z}_+^q \times \mathbb{R}_+^{m-q}\}. \tag{2}$$

Note that the variables x have been replaced with $\hat{x}$, which signify that the values of the first stage variables are provided to the subproblem as an input.

The master problem is a reformulation of the original problem (1), given by

$$\min_{x,\varphi}\{c^\top x + \varphi \,|\, Ax \geq b, \theta(x) - \mathbf{1}\varphi \leq \mathbf{0}, \phi(x) \leq \mathbf{0}, x \in \mathbb{Z}_+^p \times \mathbb{R}_+^{n-p}, \varphi \in \mathbb{R}\}. \tag{3}$$

The variable φ is an underestimator of the optimal objective value of the subproblem (2) and $\theta : x \in \mathbb{R}^n \to \mathbb{R}^o$ and $\phi : x \in \mathbb{R}^n \to \mathbb{R}^f$ denote the constraint functions for the *optimality* and *feasibility* cuts respectively. The underestimation of the subproblem objective is given by the optimality cuts $\theta(x) - \mathbf{1}\varphi \leq \mathbf{0}$. Additionally, the feasibility cuts $\phi(x) \leq \mathbf{0}$ are provided to eliminate any solutions $\hat{x}$ that induce infeasible instances of problem (2). In the classical description of BD, both $\theta(x)$ and $\phi(x)$ are linear functions; however, in the general application of the decomposition technique, they need not be [23]. The optimality and feasibility cut functions are used to describe numerous cut types that support the decomposition of many classes of MIPs and CIPs. In the application of BD to problem (1), linear optimality and feasibility cuts are sufficient to find the optimal solution. The *optimality* and *feasibility* cuts are collectively described as *BD cuts* throughout.

The sets of BD cuts typically have a cardinality that is significantly larger than the number of original second stage constraints. In many instances of problem (3), this increase in constraints makes the problem intractable. BD effectively handles this explosion in the number of constraints describing problem (3) by employing a delayed constraint generation algorithm. The BD algorithm relaxes problem (3) by replacing constraints $\theta(x) - \mathbf{1}\varphi \leq \mathbf{0}$ and $\phi(x) \leq \mathbf{0}$ with only a subset of these constraints, denoted by $\hat{\theta}(x) - \mathbf{1}\varphi \leq \mathbf{0}$ and $\hat{\phi}(x) \leq \mathbf{0}$ respectively. This relaxed problem is labelled as the BD master problem.

1.1.1 Traditional BD algorithm

The traditional approach for solving problem (1) by BD involves an iterative algorithm. This algorithm consists of two major steps: solving the master problem to identify candidate solutions, followed by solving the subproblem to generate cuts. More specifically, the first step solves the master problem to compute a lower bound on the optimal objective function value of problem (1), given by $LB = c^\top \hat{x} + \hat{\varphi}$. Given the solution $(\hat{x}, \hat{\varphi})$ from the master problem, the second step involves verifying the solution with respect to the original problem constraints using Algorithm 1. If primal and/or dual solutions are returned from Algorithm 1, then an *optimality* or *feasibility* cut should be generated and added to the

Algorithm 1: BD subproblem solving

Data: Master problem solution $(\hat{x}, \hat{\varphi})$ and Subproblem (2)

Result: Feasibility of (2), an upper bound UB for the master problem and, if available, primal and dual solutions or a dual ray from (2)

```
begin
    given (x̂, φ̂), solve (2) to optimality
    switch solution to (2) do
        case i) (2) is infeasible do
            return INFEASIBLE, UB ← ∞ and, if (2) is convex, a dual ray of (2)
        case ii) (2) is feasible ∧ z(x̂) > φ̂ do
            return FEASIBLE, UB ← c^⊤ x̂ + z(x̂) and primal and, if (2) is convex, dual solutions of (2)
        case iii) (2) is feasible ∧ z(x̂) ≤ φ̂ do
            return FEASIBLE and UB ← c^⊤ x̂ + z(x̂)
```

master problem. If a cut is generated, then the master problem is resolved to identify a new candidate solution. If no cut is generated or the difference between LB and UB is within a given tolerance, the BD algorithm terminates. The optimal solution for problem (1) is given by $(\hat{x}, \hat{y}(\hat{x}))$—where $\hat{y}(\hat{x})$ is the feasible extension of $\hat{x}$ and is found by solving problem (2) using $\hat{x}$ as input—with an objective value of $c^\top \hat{x} + z(\hat{x})$.

1.1.2 Branch-and-cut approach

While the traditional approach for applying the BD algorithm may appear the most intuitive, there are a number of limitations to its use. First, it must be stated that if problem (1) is an LP, there is no difference between applying the traditional or branch-and-cut approach. The main limitations arise when problem (1) consists of discrete variables. In the traditional algorithm, only the optimal solution to the master problem is provided to Algorithm 1 for verification. The first issue is that this approach ignores the many integer feasible solutions—which are sub-optimal for the current iteration but potentially optimal for the original problem—that are encountered during the branch-and-bound search. Second, building and discarding a complete branch-and-bound tree introduces a large amount of overhead, especially since many executions of Algorithm 1 are typically required.

Since the reformulation of problem (1) as a result of applying BD consists of an exponential number of constraints, a branch-and-cut approach is one of the most appropriate methods for handling this prohibitively large problem. The branch-and-cut approach for implementing the BD algorithm takes advantage of the callback functions that are available within modern mixed integer programming (MIP) and constraint integer programming (CIP) solvers. The concept of the branch-and-cut approach to BD was first introduced by Geoffrion and Graves [24]. This approach, under the name of branch-and-check, was discussed in detail by Thorsteinsson [54] as a hybrid of MIP and constraint programming. In the context of MIP, Fortz and Poss [19] presents a more in-depth study of the branch-and-cut approach for BD. As highlighted by Rahmaniani et al. [50], there are many applications of BD using the branch-and-cut approach; however, there is still many open questions about the best use of this algorithmic framework for BD.

A branch-and-cut algorithm involves solving a relaxation at nodes of the branch-and-bound tree, whose solutions are verified by cut generation algorithms, such as Algorithm 1, to ensure the relaxation solutions are valid for the original problem. If the relaxation solution is found to violate any omitted constraints, then valid cuts must be added and the relaxation is resolved. In the context of BD, all integer feasible solutions $(\hat{x}, \hat{\varphi})$ found when solving the LP relaxation of nodes during the tree search must be verified by executing Algorithm 1. The returned primal and dual solutions are then used to generate globally valid cuts for problem (1). The upper bound UB returned from Algorithm 1 is a global upper bound that can be used to prune nodes in the branch-and-cut tree. If UB is equal to the best known lower bound for problem (1), then $\hat{x}$ is optimal for the original problem. The optimal solution to problem (1), given by $(\hat{x}, \hat{y}(\hat{x}))$, is found by solving problem (2) with the solution $\hat{x}$ that is identified to provide the best upper bound.

The main point of difference between the traditional and branch-and-cut approach for BD arises if problem (1) contains discrete variables. In this case, the former solves a discrete master problem to integer optimality multiple times where the latter only once. An advantage of the branch-and-cut approach is that all integer feasible solutions encountered during the branch-and-bound search are verified by Algorithm 1. This removes some of the overhead from repeatedly generating branch-and-bound trees in the traditional approach; however, this comes at the cost of potentially more executions of Algorithm 1.

The BD framework presented in this paper employs the branch-and-cut approach. Throughout, when referring to the BD algorithm, this is in reference to the approach described in this section.

Please note, that while φ is presented as single dimensional in problem (3), if D exhibits a block diagonal structure the subproblems can be disaggregated and a single φ is defined for each subproblem. In the proposed framework, the subproblems are always disaggregated. However, only a single dimensional φ is used throughout to simplify the discussion.

1.2 Related work

The suitability of decomposition methods for solving structured problems has led to a rise in software methods to provide some level of automation in the application of decomposition techniques. At the most basic level, automated decomposition software methods provide the core functionality of a particular decomposition algorithm, such as the pricing loop for column generation or the cutting loop for BD, and then require the user to supply the master and subproblems in a suitable form for the software. The most sophisticated software solutions involves automatic detection of structure in the constraint matrix and the application of decomposition methods and algorithms. Unfortunately, few, if any, software packages provide the complete spectrum from basic core functionality through to complete automation. The BD framework presented in this paper fills this gap by providing implementation options covering the broad range of automation methods.

A common feature of modern mathematical programming solvers is the ability to interact with the branch-and-bound algorithm through various different callback functions. Such callback functions are used to incorporate solving features such as primal heuristics, separation routines or presolving methods and custom algorithms, including column generation and the BD algorithm. The branch-and-cut approach for BD is implemented using callback functions that are available for checking violated inequalities from the LP solution and checking the feasibility of primal solutions. Thus, the basic functionality for the implementation of BD is available within many mathematical programming solvers, such as ABACUS [31], ALPS [57], BCP [33, 52], CPLEX [30], Gurobi [26], MINTO [45], SCIP [25], SYMPHONY [51] and Xpress [17].

While modern mathematical programming solvers provide the flexibility to implement the BD algorithm, much effort is still required from the researcher or developer. In particular, all features and enhancements that are typical of a BD algorithm need to be implemented, most commonly in a problem specific manner.

In recent times, software packages have emerged to simplify the application of BD. A major source of such software packages has been the stochastic programming community. One of the earliest software packages dedicated to solving stochastic programming problems is FortSP [58]. FortSP provides various algorithms for solving linear stochastic programming problems, including variants of BD. As an alternative to BD, dual decomposition is commonly applied to solve stochastic programming problems, especially when the first and second stage problems include some discrete variables. A very basic package providing the dual decomposition algorithm is ddsip [40], which uses the ConicBundle package [28] for updating the Lagrangian dual multipliers and CPLEX for solving LP relaxations at each node in the branch-and-bound tree. Interestingly, ddsip is only capable of handling stochastic programs consisting of discrete variables, and not purely continuous problems. An open-source package for stochastic programming based on Python is PySP [56]. This package provides functionality for formulating and solving stochastic programs, including options for using the progressive hedging or BD algorithms. A general implementation of the dual decomposition method is given in DSP [32] and is capable of using the underlying sequential solvers of CPLEX and SCIP. A BD implementation is also available in DSP for solving stochastic programming problems; however, this feature is only compatible with SCIP 3.2 [22]. The PIPS solvers have been extended to include a large-scale parallel dual decomposition solver [35] that has been successfully applied to solve many instances from the SIPLIB [2]. Finally, CPLEX [30] have redesigned the callback functions and introduced a BD framework. Since version 12.6, dedicated callback functions exist for implementing BD, and the core functionality of the algorithm is provided internally to the solver.

Three levels of implementation flexibility is typically provided by decomposition software packages. The first level is characterised by the *lack* of flexibility in the solution methods. Specifically, the complete decomposition algorithm is automated and no input from the user is required except for a problem instance supplied in a standard format such as the SMPS [10] format. The second level of flexibility allows some interaction with the decomposition algorithm through the use of modelling tools. This could be in the form of algebraic modelling languages such as Pyomo [27] and StochJuMP [29], which form the basis of the modelling tools available within PySP and DSP respectively. When using modelling tools, the user specifies mathematical programs for the master and subproblems that will then be used within the decomposition algorithms. The use of modelling tools, compared to using standard instance formats, provides the capability to employ custom master and subproblem solvers within the decomposition algorithm. A further level of implementation flexibility is through the use of callback functions and solver interface methods. As an example, PySP provides callback methods that are executed at various stages of the progressive hedging algorithm. These callback methods are provided primarily to facilitate the implementation of various enhancement methods that are typically required to achieve fast convergence of the algorithm. To support the use of custom solvers in the dual decomposition and BD algorithms, DSP defines a solver interface base class that can be used to derive classes for solving the MIP and LP master and subproblem instances.

There has been growth in the number of software packages that perform automatic detection and decomposition for general MIP instances. Prominent examples include BaPCod [55], DIP [21], G12 [49]

and GCG [8]. The key feature of automated decomposition packages is the detection of structure within the constraint matrix. Permuting the columns and rows of the constraint matrix can expose structure that is suitable for the application of decomposition techniques. The primary foci of BaPCod, DIP, G12 and GCG are the decomposition methods of Dantzig-Wolfe reformulation and Lagrangian relaxation, and hence the detected structures are not always suitable for the application of BD. This is because both Dantzig-Wolfe reformulation and Lagrangian relaxation are best applied to problems exhibiting linking constraints, whereas BD requires a set of linking variables. Recently, Bastubbe [6] extended the detection schemes within GCG to detect problem structures that are more suitable to the application of BD. To the best of the authors knowledge, the work of Bastubbe [6] is the only example of structure detection for BD. This paper will detail the extensions to GCG that enable the use of this structure detection to apply BD for solving general MIPs.

1.3 Contributions and paper structure

The overarching goal of this work is to simplify the application of BD for solving general MIPs and CIPs. The main contributions of this work are:

- The development of a general BD framework within the state-of-the-art CIP solver SCIP.
- A plugin-based design that provides flexibility in the application of the BD framework to solve a wide range of real-world applications. This flexibility includes the ability to insert custom cut generation methods and the possibility to define custom solving methods for the BD subproblems.
- The development of a high performance BD algorithm with the inclusion of numerous classical and novel enhancement techniques.
- A framework that can be used for automatic structure detection and application of BD.
- An extensive computational study to evaluate the BD framework and the available enhancement techniques. This computational study is one of the only studies that evaluates such a wide range of enhancement techniques on a collection of different problem types using the same computational and software platform.

The structure of this paper is as follows: A detailed overview of the BD framework is provided in Sections 2 and 3. Section 2 discusses the algorithmic methodology with reference to the SCIP solving process, where Section 3 presents the key implementation details for the BD framework. The main features of the BD framework are presented in Section 4, including a description of the available classical and novel enhancement techniques. An important feature of the BD framework is the compatibility with the automatic branch-price-and-cut solver GCG. The extension of GCG to automatically apply BD to general MIP instances is described in Section 5. An extensive computational study evaluating a number of available enhancement techniques is presented in Section 6.

The BD framework described in this paper is part of the SCIP Optimization Suite. The source code for this framework, example applications and relevant documentation can be found at `scip.zib.de`.

2 Benders' decomposition framework

The BD framework is a new core feature of the SCIP Optimisation Suite that was first released in version 6.0 [25]. For details regarding the software design of SCIP the reader is referred to the PhD thesis of Achterberg [1]. The main features of the BD framework is the design of two new plugins—the Benders' decomposition and cut generation plugins—and the implementation of the core solving process of the BD algorithm.

The framework is designed to handle decompositions of the form:

- mixed integer master problem and continuous subproblem,
- binary master problem and constraint integer program subproblem.

Additionally, the plugin-based structure provides sufficient flexibility to extend this framework to handle other problem classes. This flexibility also enables the users of SCIP to employ the BD algorithm in many different ways. First, instances available in the SMPS format [10] can be supplied directly to SCIP and solved by BD. Second, a user can employ the default BD algorithm using the SCIP callable libraries when building a problem-specific solver. Third, a custom BD algorithm, with alternative solving and cut generation methods, can be implemented using the plugin-based structure of the framework. These three methods for using the framework are detailed in Section 4.1.2. Finally, through GCG, automated structure detection and decomposition can be applied to solve general MIP problems by BD (Section 5).

The presented discussion will strive to achieve two main goals. The first is to describe the implementation details of the BD framework that is available in SCIP since version 6.0. The second is to help guide future researchers in the implementation of BD algorithms, either general frameworks or problem specific implementations, in modern general purpose MIP and CIP solvers.

2.1 The SCIP solving process and Benders' decomposition

The branch-and-cut algorithm of modern MIP and CIP solvers can be described as a highly sophisticated tree search algorithm. The sophistication of the branch-and-cut algorithm comes from the many semi-autonomous component algorithms, such as presolving methods, primal heuristics and propagation routines, that are managed by a central control mechanism. Control mechanisms orchestrate the various component algorithms, by executing them at different stages during the processing of a node. The control mechanism, or solving process, for SCIP is presented in Figure 1. A fundamental part of processing a node is the *Node Processing* stage. Primarily, this stage solves a bounding problem, typically the LP relaxation of the original problem augmented with branching constraints and valid inequalities, and generates further valid inequalities as required. Additionally, delayed constraint generation methods, such as BD, are executed during the *Node Processing* stage if the LP solution violates any omitted constraints. Outside of *Node Processing*, if the LP relaxation violates integrality restrictions, the *Branching* is performed. This is followed by the execution of *Primal Heuristics* to identify primal feasible solutions that can improve the global upper bound. Other process are performed during the processing of a

node, such as *Domain propagation* and *Conflict analysis*, however they are not important for the work discussed in this paper.

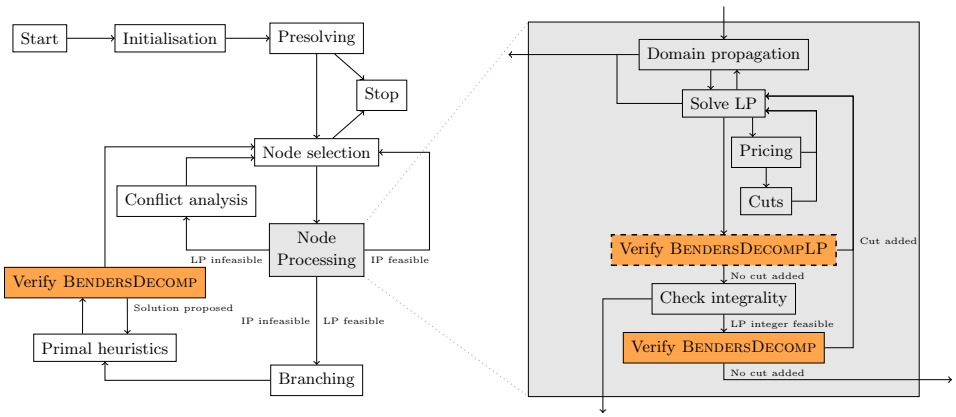


Figure 1: The solving process of SCIP including the verification of the BD constraint handlers.

Drawing upon concepts from constraint programming, delayed constraint generation methods can be described as the verification of arbitrary constraints. These arbitrary constraints receive as input a solution from the LP relaxation at a node in the branch-and-cut tree that is then verified for feasibility with respect to the omitted original problem constraints. If the verification results as False, then constraints, and potentially additional variables, must be added to the relaxed original problem at that node, or globally throughout the tree. If the verification results as True, then the input solution is reported as feasible with respect to the omitted constraints.

The delayed constraint generation method of BD can be described as such an arbitrary constraint. The BD constraint, given by

$$\text{BendersDecomp}(x, \varphi) \Leftrightarrow \text{problem (2) is feasible and } z(x) \leq \varphi, \tag{4}$$

takes the input solution $(\hat{x}, \hat{\varphi})$ and verifies feasibility with respect to the subproblem constraints by executing Algorithm 1. If cases i) or ii) occur when executing Algorithm 1, then BendersDecomp verifies to False and the feasible region of the LP relaxation at the node must be reduced through the addition of constraints to eliminate the solution $(\hat{x}, \hat{\varphi})$. If case iii) of Algorithm 1 occurs, then BendersDecomp verifies to True.

Figure 1 shows how the verification of BendersDecomp is integrated into the solving process of SCIP when solving the master problem. This verification of BendersDecomp is performed in two separate places, a) after the checking the integrality of the LP solution, called *constraint enforcement* and b) after candidate solutions are found by primal heuristics, called *solution checking*. Considering a), if, given an integer LP solution, BendersDecomp is verified to False, then an optimality or feasibility cut is added to the LP. This addition of BD cuts triggers a resolve of the bounding problem. Otherwise, if BendersDecomp verifies to True, the processing of the node ends and the next node is selected. In case b), a primal solution is only feasible for the original problem if BendersDecomp evaluates to

TRUE. In this case, no optimality or feasibility cuts need to be generated from the returned primal and dual solutions or rays. If case ii) of Algorithm 1 occurs, then $(\hat{x}, \hat{\varphi})$ is not feasible for the master problem; however, $(\hat{x}, z(\hat{x}))$ is feasible. Thus, an alternative primal feasible solution is proposed for addition to the solution pool via the available primal heuristics.

An optional extension to SCIPs solving process is the verification of BENDERSDECOMPLP. If problem (3) contains discrete variables and problem (2) is convex, or a convex relaxation of the CIP exists, then this verification supplies fractional solutions from the LP to Algorithm 1. After solving the convex relaxation of problem (2), if case i) or ii) occur, then the returned dual solution or rays are used to generate optimality or feasibility cuts respectively. This additional step will be discussed in Section 2.2.2 when describing the classical enhancement technique of the three-phase method.

2.2 Constraint handlers

The BENDERSDECOMP and BENDERSDECOMPLP constraints are implemented as constraint handlers within SCIP. These constraint handlers provide an interface between the SCIP solving loop and the BD core functionality, which will be described in Section 2.3.

2.2.1 Fundamental verification stages

The two stages at which BENDERSDECOMP is verified (shown in Figure 1) enforce the constraints of the LP relaxation and check the feasibility of candidate primal solutions. Similar to many modern general purpose solvers, these two stages are accessed through callback functions. The enforcement of constraints for BENDERSDECOMP is performed in the callback function `conshdlrEnfolpBenders` and the solution checking is performed in `conshdlrCheckBenders`. These callback functions are similar to the cut callback functions that are available within solvers such as CPLEX, GUROBI and XPRESS. In both callback functions, Algorithm 1 is executed; however, in the latter no BD cuts are generated from the returned dual solutions or rays. The implementation of BD in SCIP requires more fine grained return results than the three cases in Algorithm 1. Specifically, for the `conshdlrEnfolpBenders` verification, the addition of a cut following case i) or ii) requires the result of CONSTRAINT ADDED or SEPARATED to be returned, the former used if a constraint is added to SCIP and the latter if a cut is added to the cut pool. For the `conshdlrCheckBenders` verification, cases i) and ii) both require a result of INFEASIBLE to be returned; however, in the latter case an alternative primal solution $(\hat{x}, z(\hat{x}))$ is proposed using the TRYSOL heuristic[1]. Proposing this alternative solution is necessary for reporting the upper bound found by executing Algorithm 1 to the central control mechanism of SCIP. Finally, for both constraint handler callback functions case iii) returns a result of FEASIBLE.

[1]For specific details regarding the process of proposing alternative primal solutions, the reader is referred to the source code of `cons_benders.c` and `heur_trysol.c` in the SCIP distribution.

2.2.2 Three-phase method

The three-phase method forms the core of the algorithmic structure of the BD framework within SCIP. This algorithm was first proposed by McDaniel and Devine [41], in the form of the two-phase method, as an enhancement technique for the BD algorithm when solving MIPs. As an extension to the two-phase method, a third phase was added by Cordeau et al. [14] to handle mixed integer subproblems in the BD algorithm.

While the three-phase method is based on the traditional BD algorithm, this algorithm is easily adapted to the branch-and-cut approach. This adaptation is achieved through the introduction of the BendersDecompLP constraint, as shown in Figure 1, which has been implemented in a constraint handler separate to the BendersDecomp constraint. The verification of BendersDecompLP at the root node is equivalent to the first phase of the three-phase method—the LP relaxation of the master problem is repeatedly solved until BendersDecompLP verifies to True. This phase is optional and is invoked by activating the BendersDecompLP constraint. The second and third phases are implemented directly within the core functionality of the BD framework. These phases are necessary for the exactness of the BD algorithm. For continuous subproblems, the second phase is performed by verifying BendersDecomp at nodes where the LP relaxation satisfies the integer restrictions. For subproblems containing discrete variables, the second and third phases are adopted from the technique of Laporte and Louveaux [34], and similarly Angulo et al. [4]. This approach involves executing Algorithm 1 for integer $(\hat{x}, \hat{\varphi})$ solutions by solving only a convex relaxation of problem (2) to generate cuts (Phase 2). When no cuts are generated from the solution to the convex relaxation of problem (2), the corresponding CIP is then solved and the appropriate cuts are generated (Phase 3). Enhancements to the three-phase method that are included in the BD framework are described in Section 4.2.

2.3 Core functionality

The core functionality of the BD framework is presented in Algorithm 2. The major components of the core functionality are the setting up of the subproblems, determining when to solve the convex relaxation of general CIP subproblems and returning the correct result for the verification type, either constraint enforcement or primal solution checking. Most importantly, the BD core determines what cuts should be generated, with respect to the problem class, and at which points in the SCIP solving process they should be generated, i.e. whether during the LP enforcement or primal solution checking.

The mechanisms for setting up and solving the subproblems are some of the most critical parts of the BD algorithm. Hidden within Line 2 of Algorithm 1 are the details for setting up problem (2) given $\hat{x}$. While the classical description of the BD algorithm involves modifying the right-hand side of the primal subproblem constraints, or equivalently the objective coefficients of the dual subproblem, this is not practical from an implementation point of view. In the BD framework, the primal subproblems are setup by fixing the corresponding master problem variables with respect to $\hat{x}$. Practically, this is facilitated by including a representative variable in the subproblem for each master problem variable present in the constraints of problem (2).

Remark 1. *The algorithms of the BD framework assume that the subproblems are given in their primal form.*

As stated in Section 2.2.2, Phases 2 and 3 of the three-phase algorithm are performed directly within the core of the BD framework. Specifically, Phases 2 and 3 start at line 3 and 10 in Algorithm 2 respectively. It is important to note that when verifying BendersDecompLP only Phase 2 is executed during Algorithm 2, while both phases are potentially executed when verifying BendersDecomp.

Algorithm 2: The core functionality of the BD framework

Data: Master problem solution $\hat{x}$, the verification *Type*
Result: The *Result* from the verification

```
1  begin
2      CutAdded ← False
       /* iterate over all subproblems of the decomposition                    */
3      using a convex relaxation of problem (2), execute Algorithm 1   // dual solutions can be returned
       /* once all convex relaxations are solved, then generate cuts           */
4      if Type is ENFORCELP then
5          if z(x̂) > φ̂ ∨ problem (2) is infeasible then
               /* classical BD cuts                                            */
6              generate a BD cut using the dual solution or dual ray of the relaxation of problem (2)
7              CutAdded ← True
8      else if Type is CHECK ∧ problem (2) is infeasible then
9          return Infeasible
       /* if there exists at least one subproblem that is a CIP                */
10     if verifying BendersDecomp ∧ CutAdded is False ∧ problem (2) is a CIP then
           /* iterate over all non-convex subproblems of the decomposition     */
11         using the general CIP problem (2), execute Algorithm 1   // no dual solutions can be returned
           /* once all non-convex subproblems are solved, then generate cuts   */
12         if Type is ENFORCELP then
13             if z(x̂) > φ̂ ∨ problem (2) is infeasible then
                   /* non-classical BD cuts, i.e.  integer, no good, logic-based, ...  */
14                 generate a BD cut using the primal solution of problem (2)
15                 CutAdded ← True
16         else if Type is CHECK ∧ problem (2) is infeasible then
17             return Infeasible
       /* returning the correct result                                         */
18     if CutAdded is True then
19         return CutAdded
20     else
21         if Type is CHECK ∧ z(x̂) > φ̂ then
22             return AuxVarViolation
23         else
24             return Feasible
```

Importantly, if at least one cut is generated when evaluating the convex (or convex relaxation) subproblems during the enforcement of LP solutions, or at least one subproblem is infeasible when checking a primal feasible solution, then the CIP subproblems are not solved. In each stage all subproblems are solved prior to the generation of cuts. This separation between solving and cut generation is motivated by the potential parallelisation of the BD algorithm. Typically the solving of subproblems is much slower than cut generation and the former is separable whereas the latter reads from and writes to shared memory.

Technical Note 1 (Internal subproblem solving methods)**.** *For performance reasons, different solving methods are used in each of the solving stages. Since the convex relaxations are typically an LP or NLP, then directly solving the* SCIP *instance of the subproblem by calling* `SCIPsolve` *would introduce too much overhead. Thus, during an initialisation step an event handler is used to interrupt the solve of the subproblem immediately after the LP or NLP is constructed—providing direct access to the internal LP or NLP. Then solving the subproblem in the first stage is achieved by entering probing mode, by calling* `SCIPbeginProbing`*, just prior to setting up the subproblem (by fixing x to $\hat{x}$) and solving the LP, by calling* `SCIPsolveProbingLP`*. The NLP is solved by calling* `SCIPsolveNLP`*. The LP, or NLP, is reverted to its unsetup state by calling* `SCIPendProbing`*.*

Solving the CIP of the subproblem employs the standard SCIP *solving methods, i.e.* `SCIPsolve`*. An event handler is still used during an initialisation phase to interrupt the solve; however, this is to ensure that the original problem has been transformed prior to executing the solving process (for information on the original and transformed problems the reader is referred to [59]). After setting up the problem, the subproblem is solved by calling* `SCIPsolve`*. The CIP is unsetup by freeing the transformed problem with a call to* `SCIPfreeTransform`*.*

2.4 Available Benders' cut methods

Four BD cut methods are available within the BD framework as part of SCIP Optimisation Suite 6.0. These methods are included to support the application of BD to decompositions of the forms stated in Section 2. In the following, each of the cut generation methods are based on the primal form of problem (2). The vector of dual solutions related to constraint set $Dy \geq g - B\hat{x}$ is denoted by π.

Classical feasibility cut This set of BD cuts are generated only if the LP relaxation of problem (2) is infeasible. In this case, the dual vector $\hat{\pi}$ comprises the Farkas dual values, which is commonly termed the dual ray. Given the dual ray $\hat{\pi}$ from an infeasible problem (2), the feasibility cut that eliminates $\hat{x}$ from problem (3) is given by

$$0 \geq \hat{\pi}^\top (g - Bx). \tag{5}$$

Classical optimality cut An optimality cut is generated from an optimal solution to problem (2). Since the optimality cut provides an underestimation of the optimal objective value of the subproblem, an auxiliary variable φ is required. Given an optimal dual solution $\hat{\pi}$ to problem (2), the optimality cut

that provides an underestimation of $z(\hat{x})$ in the master problem is given by

$$\varphi \geq \hat{\pi}^\top (g - Bx). \tag{6}$$

No-good cut The first of the CIP BD cut methods is employed to eliminate master solutions that induce infeasible instances of problem (2). This cut is only applicable when the master problem is pure binary; the subproblem can be a general CIP, but with at least one discrete variable.

Let $\mathcal{B}$ be the index set of all binary variables in the master problem, which in this case is all variables in the vector x. Given the master solution $\hat{x}$, let $\mathcal{B}^+ = \{i \,|\, \hat{x}_i = 1, i \in \mathcal{B}\}$ be the index set of master variables that take a solution value of 1. If problem (2) is evaluated to be infeasible, then $\hat{x}$ is eliminated from the master problem with the addition of the constraint

$$\sum_{i \in \mathcal{B}^+} (1 - x_i) + \sum_{i \in \mathcal{B} \setminus \mathcal{B}^+} x_i \geq 1. \tag{7}$$

The no-good cut only eliminates a single solution from the master problem.

The combinatorial Benders' cut proposed by Codato and Fischetti [13] are a stronger version of the classical feasibility and no-good cuts. This strengthening is achieved by defining a set $\mathcal{B}' \subseteq \mathcal{B}$ and then generating a no-good cut using this set to eliminate master problem solutions. However, the generation of this type of cut typically relies on specific structures, commonly indicator constraints, for deducing combinatorial arguments to identify $\mathcal{B}$ and eliminate master solutions. The development of a general method for generating these cuts is a point for future research.

Laporte and Louveaux integer cut This BD cut is an analogue of the classical optimality cut for continuous subproblems. The Laporte and Louveaux cut [34] provides a point-wise underestimation of the objective function to problem (2) at $\hat{x}$. Similar to the no-good cut, this cut generation method can only be applied when the master problem is pure binary, while the subproblem can be a general CIP with at least one discrete variable.

Given a solution to the master problem $\hat{x}$ that induces a feasible instance of problem (2), define the sets $\mathcal{B}$ and $\mathcal{B}^+$ as above. Let $\hat{z}(\hat{x})$ denote the optimal objective value of problem (2) given $\hat{x}$. Also, let L be a valid lower bound of problem (2) without any fixing of x. The cut that provides an underestimation of problem (2) at $\hat{x}$ is given by

$$\varphi \geq \big(\hat{z}(\hat{x}) - L\big)\left(\sum_{i \in \mathcal{B}^+} (x_i - 1) - \sum_{i \in \mathcal{B} \setminus \mathcal{B}^+} x_i + 1 \right) + L. \tag{8}$$

The interpretation of constraint (8) is the following: At $\hat{x}$ the right hand side of (8) is equal to $\hat{z}(\hat{x})$. Thus, if $(\hat{x}, \hat{\varphi})$ is the optimal solution to problem (3), then $\hat{\varphi} = \hat{z}(\hat{x})$ is the optimal underestimator of problem (2). For each unit difference in the binary vector x from the solution $\hat{x}$, the right hand side of (8) decreases by $(\hat{z}(\hat{x}) - L)$ units. As such, the strength of (8) is directly related to the quality of the lower bound L. The value of L can be updated during the solution process as better estimates of the lower bound of $z(x)$ are found.

3 Extensions to the SCIP plugin structure

The plugin structure of SCIP has been extended with two additional plugins for the BD framework. These plugins provide an interface for users to extend the framework with alternative subproblem solving methods or novel cut generation techniques.

3.1 Benders' decomposition plugin type

The BD plugin provides users with the fundamental functionality for interacting with the BD framework during the execution of Algorithm 1. Primarily, this plugin type is used to create the BD subproblems and provide a mapping between the master and subproblem variables. In addition, flexibility in the implementation of the BD algorithm is afforded through the option to define custom subproblem solving routines and auxiliary enhancement algorithms.

Throughout the following sections, the callback function names will be given with a suffix `Xyz`. In the actual implementation of these functions `Xyz` should be replaced with a name corresponding to the plugin name, such as `Default` for the DEFAULT BD plugin.

3.1.1 Fundamental callback functions

There are two fundamental callback functions of a custom BD plugin: `bendersCreatesubXyz` and `bendersGetvarXyz`. Only specifying the fundamental callback functions informs the BD core that the internal solving methods, which are described in Technical Note 1, must be used. Thus, a convex relaxation of a CIP will be formed so that the two solving and cut generation stages of Algorithm 2 can be executed.

Creating and registering the subproblems The main purpose of the callback function `bendersCreatesubXyz` is to provide an appropriate place to call `SCIPaddBendersSubproblem` for each subproblem. A call to this function, with the SCIP pointer to the respective subproblem, increases the subproblem counter in the BD core and stores the pointer to the subproblem in the internal data structures. Contrary to its name, it is not necessary to actually create the SCIP instance of the subproblem during the call to this callback function: it may be created earlier during the program execution, such as while reading the problem data.

If a custom subproblem solving method is desired, then it is not necessary to provide a SCIP pointer to `SCIPaddBendersSubproblem`; however, it is necessary to inform the BD core that a subproblem exists. In this case, a `NULL` pointer must be provided to `SCIPaddBendersSubproblem` in place of the subproblem SCIP pointer.

Variable mapping The mapping between the master and subproblem variables is a critical component of the BD framework. This mapping is used to identify the master problem variables in the subproblem corresponding to $\hat{x}$ and in the cut generation process. Both of these key processes are facilitated through the variable mapping callback function `bendersGetVarXyz`.

The parameters of the `bendersGetvarXyz` callback function are:

- `var`: the variable for which a corresponding master or subproblem variable is desired.
- `probnumber`: the index of the subproblem where the mapped variable is located, or -1 for mapping from the subproblem to the master problem variables.

Given these input parameters, the callback function must set the pointer `mappedvar` to the variable contained in subproblem `probnumber` (or master problem if -1) corresponding to the input variable `var`. During the execution of Algorithm 2, the variable mapping is accessed through the public API functions `SCIPgetBendersMasterVar` and `SCIPgetBendersSubproblemVar`.

Technical Note 2 (Creating and storing the variable mapping)**.** *Given its importance to the BD framework, users must take care when specifying the variable mapping. In the* DEFAULT *BD plugin (described in Section 3.1.3) that is available within* SCIP*, the variable mapping is created using the variable names. When using the* DEFAULT *BD plugin, it is required that for each master problem variable the same name is used in the master and subproblem* SCIP *instances. By comparing the variable names of the master and subproblem variables, a hashmap is created during the initialisation process so that the appropriate mapped variable can be retrieved with a call to `SCIPgetBendersMasterVar` or `SCIPgetBendersSubproblemVar`.*

As opposed to using the variable names to create the variable mapping, a user may create a hashmap when building the master and subproblem SCIP *instances. This hashmap can be provided to the custom BD plugin as a parameter to the decomposition creation function.*

One must consider the original and transformed variables when defining the BD variable mapping (see [59] for more information about the original and transformed problem). A mapped variable may be requested for either an original or transformed variable during the solution process. Thus, it is important to ensure that the variable mapping is suitable for both the original and transformed problems.

3.1.2 Optional callback functions

Customisation of the BD algorithm is achieved by the implementation of optional callback functions in the BD plugin. Within Algorithm 1, callback functions are provided to replace the internal subproblem solving methods of the BD framework with custom methods. In addition, it is possible to interrupt the verification of BENDERSDECOMP and BENDERSDECOMPLP immediately prior to and after the execution of Algorithm 2.

Custom solving functions For particular applications custom solving methods may be more effective than the methods available within the BD framework, i.e. those described in Technical Note 1. To enable the specification of custom solving methods two optional callbacks have been provided: `bendersSolvesubconvexXyz` and `bendersSolvesubXyz`. The former is called at line 3 in Algorithm 2 to solve convex, and convex relaxations of, subproblems, and the latter is called at line 11 in Algorithm 2 to solve CIP subproblems. If either `bendersSolvesubconvexXyz` and `bendersSolvesubXyz` are specified,

then a method for freeing the subproblem between verifications of BENDERSDECOMP and BENDERSDECOMPLP is required. For example, if the SCIP instance is solved directly, i.e. by calling `SCIPsolve`, then the user must call `SCIPfreeTransform` to unsetup the variable fixings made with respect to $\hat{x}$. This freeing process is performed during the execution of the optional callback function `bendersFreesubXyz`.

Technical Note 3 (Setup subproblem for custom solving method)**.** *If the internal solving methods are used, the subproblems are automatically setup with respect to the input solution $\hat{x}$ using the supplied variable mapping. Similarly, the subproblems are automatically unsetup after the completion of Algorithm 2. If custom solving callback functions are specified, then the subproblems must be setup by the user prior to executing the custom solving method. This must be performed in `bendersSolvesubconvexXyz` and `bendersSolvesubXyz`. A `SCIP_SOL` pointer to the master problem solution $(\hat{x}, \hat{\varphi})$ to be verified is provided to the solving callback functions and can be used to set up the subproblems. If the subproblems are specified as* SCIP *instances, the setup can be achieved by calling `SCIPsetupBendersSubproblem`. Otherwise, the appropriate subproblem variables can be fixed to $\hat{x}$ by identifying x using `SCIPgetBendersMasterVar`.*

Note that `bendersFreesubXyz` for unsetting up the subproblems is called at the end of Algorithm 2. Thus, if a subproblem is solved in both `bendersSolvesubconvexXyz` and `bendersSolvesubXyz`, then the subproblem must be set up such that it does not require a call to `bendersFreesubXyz` between solving phases. For the internal solving methods, the CIP subproblems are transformed and setup immediately prior to solving the convex relaxation. They are then freed after solving the CIP by calling `SCIPfreeTransform` or `SCIPendProbing` if the subproblem is convex.

Pre- and post-solving methods There may be circumstances where it is desired to interact with the verification of BENDERSDECOMP before and after the solving process, described in Algorithm 2, is executed. Such circumstances include inserting an alternative, possibly faster, solving method prior to the main solving process or checking the generated cuts and performing strengthening procedures. This interaction with the core functionality of the BD framework is provided through the callback functions `bendersPresubsolveXyz` and `bendersPostsubsolveXyz`. The former is called immediately before the execution of Algorithm 2 and the latter is called immediately after. Examples regarding the use of these callback functions are the cut strengthening (pre-subproblem solving) and subproblem merging (post-subproblem solving) methods described in Sections 4.5 and 4.6 respectively.

3.1.3 Default BD plugin

As part of the SCIP Optimisation Suite, a DEFAULT BD plugin is provided to facilitate the use of the framework. Only the fundamental BD callback functions are implemented within the DEFAULT plugin. As stated in Technical Note 2, the variable mapping is created by using the variable names. Since no custom solving methods are specified, the internal methods are used to solve problem (2) when executing the solving process described in Algorithm 2.

The DEFAULT BD plugin is invoked by calling the plugin creation function `SCIPcreateBendersDefault` and supplying the master SCIP instance, an array of subproblem SCIP instances and the number of

subproblems in this decomposition as parameters.

Technical Note 4 (Including and creating a custom BD plugin)**.** *To register a BD plugin with* SCIP*, it is useful to specify an include function that is called during the problem creation stage. There are a number of key steps that must be performed in the include function for the BD plugin. First, either* `SCIPincludeBendersBasic` *or* `SCIPincludeBenders` *must be called to register the custom BD plugin and corresponding callback functions with the* SCIP *core—the former only requires the fundamental callback functions to be specified where the latter requires all to be specified. Second, the BD cut plugins that will be used to generate cuts for this decomposition must be included. All default BD cut plugins can be included by calling* `SCIPincludeBendersDefaultCuts`*. Optionally, the memory for any required data structures can be allocated in the include function. All plugin specific data should be stored in* `SCIP_BENDERSDATA`*.*

While not necessary for custom SCIP *solvers, a "create" function can be implemented to add any subproblem solving data to the BD plugin during the creation of the master* SCIP *instance. A necessary step in invoking the BD framework is the activation of the BD plugins. This is achieved by calling* `SCIPactivateBenders`*, which can be executed during the BD plugin creation process.*

3.2 Benders' decomposition cut plugin type

The Benders' decomposition cut plugin type provides flexibility in the cut generation methods of the BD algorithm. Importantly, this plugin type allows for the easy extension of the framework to incorporate alternative methods for generating BD cuts. A number of classical cut generation methods are included in the form of BD cut plugins as part of the BD framework. The set BD cut generation methods available within SCIP Optimisation Suite are described in Section 2.4

The BD cut plugin has a very simple design: consisting of only a single fundamental callback, `benderscutExecXyz`, that is used to generate BD cuts. The parameters of the execution callback function are:

- `sol`: the master problem solution $(\hat{x}, \hat{\varphi})$ that has been input for the verification of BendersDecomp or BendersDecompLP.
- `probnumber`: the index of the subproblem whose solution should be used to generate a BD cut.
- `type`: the type of the verification—whether enforcing the constraints for an LP solution or checking the feasibility of a primal solution.

Using the index `probnumber`, the subproblem SCIP instance is retrieved by calling `SCIPbendersSubproblem`. The subproblem SCIP instance stores all solution information—most importantly primal and dual solutions—from the last solve executed. The solution information from the last solve can then be used to evaluate whether a BD cut needs to be generated, and, if necessary, adds it to the master SCIP instance. If a BD cut is added, then the appropriate result must be returned to the BD solving process, see Technical Note 5.

Technical Note 5 (Adding BD cuts)**.** *The BD cut generation method can add either a constraint or a cut to the master* SCIP *instance. There is an important distinction between cuts and constraints in* SCIP*, which is primarily based on when and where the inequality is generated. Specifically, constraints are added directly to* SCIP*, whereas cuts are added to the cut pool and then become a row in the LP when separation is performed on the cut pool. Whether a cut or constraint should be added from the BD cut plugin is problem and user specific. An example of how to add constraints or cuts is available in the classical optimality cut plugin* BDCutOptimality[2]*.*

Most importantly, if a constraint is added, then the result of `CONSADDED` *must be returned from* `bendersCutExecXyz`*. Alternatively, the addition of a cut requires the result* `SEPARATED` *to be returned. The result* `FEASIBLE` *is returned if no BD cuts are generated. The different results inform* SCIP *how to proceed following the verification of* BendersDecomp *or* BendersDecompLP*.*

As shown in Algorithm 2, the cut generation methods are called after all subproblems are solved, specifically lines 6 and 14. Classical BD cut plugins, i.e. those that generate cuts from the dual solutions of the subproblem, are called at line 6 and non-classical BD cut plugins, i.e. those that generate cuts from the solutions of general CIP subproblems, are called at line 14. When including the BD cut plugins, by calling `SCIPincludeBenderscut`, a parameter is set to specify whether the dual solution can be used during the execution method for generating the BD cut. A priority is also given to each of the BD cut plugins, which specifies the order in which they are called in Algorithm 2. The BD cut plugins are called in decreasing priority order until one generates a valid cut and adds it to the master problem.

The BD cut generation methods described in Section 2.4 are presented in decreasing priority order for each stage of the solving process. The classical feasibility and optimality cut generation methods are executed in the first stage and the no-good and integer cuts are executed in the second stage of Algorithm 2.

3.3 Technical notes summary

Throughout Sections 2 and 3, technical notes are provided to guide future developers of a Benders' decomposition implementation. These technical notes present important details regarding general implementations of the BD algorithm and how to interact with the BD framework of SCIP. In particular, these notes discuss:

- Technical note 1: The use of LP or NLP solvers, as opposed to MIP solvers, when solving the BD subproblem for solution verification and cut generation.
- Technical note 2: The specification of a mapping between the master and subproblems to facilitate cut generation and subproblem setup.
- Technical note 3: The use of custom solving methods for the BD subproblem.
- Technical note 4: Details on the necessary function calls for including custom BD plugins in a SCIP solver.

[2]The source code is available in `benders_opt.c`

- Technical note 5: The appropriate methods and return codes for Benders' cut addition within custom BD cut plugins.

These notes aims to share insights gained from the development of a general BD framework.

4 Framework features

An effective BD algorithm typically requires the use of various features and enhancement techniques. To improve the performance of the BD framework in SCIP, many classical enhancement techniques have been made available by default. In addition, a number of novel features have been developed for the BD framework. The details of the available enhancement techniques and the novel features of the framework will be presented in the following sections.

Many of the features are controlled through the use of run time parameters. For conciseness, we define `<x>` as `benders/<bendersname>`, where `bendersname` is the unique name given to the BD plugin that the parameter is associated with, for example `default` for the DEFAULT BD plugin.

4.1 Using the BD framework

To facilitate the broader use of the framework for custom applications, different levels of flexibility have been embedded into the software design. These levels of flexibility range from complete automation through to a custom built BD algorithm.

4.1.1 SMPS file reader

The simplest method for invoking the BD framework is by providing a stochastic programming instance to SCIP using the SMPS format [10].

One of the strengths of the SMPS format is the flexibility it affords in describing stochastic programming problems. In particular, the scenarios can be described as discrete scenarios, as blocks that combine to form scenarios or as independent distributions. Each type is identified within the input files by the section headers of `SCENARIOS`, `BLOCKS` or `INDEPENDENT`. While the SMPS format is capable of specifying a stochastic programming problem by independent distributions, only discrete distributions are supported within SCIP, i.e. those specified using the `SCENARIOS` or `BLOCKS` key words. Both two- and multi-stage problems can be provided to SCIP using the SMPS format; however, BD can only be used to solve two-stage problems.

4.1.2 BD algorithm flexibility

Complete automation Supplying a problem instance in the SMPS format, along with the setting `reading/storeader/usebenders` to TRUE, will invoke the completely automated form of the BD algorithm. All of the internal solving and BD cut generation methods will be used to solve the instance.

Complete automation of the BD algorithm can also be achieved using the DEFAULT BD plugin. When developing a custom SCIP solver, the BD framework can be invoked by calling `SCIPcreateBendersDefault` during the problem creation stage.

Custom BD cut generation methods The most versatile, and simplest, implementation of a cut generation method is to use the SCIP API to extract primal and dual information from the subproblem SCIP instances to form BD cuts. This will allow the custom cut generation method to augment all BD plugins that are available by default. If the user were to exploit more flexibility of the framework, such as a custom BD plugin, then custom cut generation method need not use the SCIP API for extracting subproblem solution information. In this setting, it is possible to share information between a custom solver and the BD cut generation method using custom API methods.

Custom BD implementation The full flexibility of the BD framework can be exploited by implementing a custom BD plugin. A custom BD implementation can be achieved in three different ways. The first method involves implementing a custom BD plugin, but only specifying the fundamental callback functions. All internal solving and cut generation methods will be employed. The second method involves defining custom solving methods for the BD subproblem; but with the subproblems defined as SCIP instances. This method of customisation provides the user with the opportunity to use custom solution algorithms for problem (2), while taking advantage of the internally available cut generation methods. The final type of customisation, while requiring much more implementation effort, exploits the full flexibility of the framework. This customisation involves implementing custom subproblem solving and cut generation methods. Thus, SCIP instances need not be created for the subproblems. If subproblem SCIP instances do not exist, then the default BD cut generation methods can not be employed. In the case, it is necessary to also implement custom BD cut generation methods.

A question that may be asked is: If the solving and cut generation methods are all custom methods, what is the advantage of using the BD framework in SCIP over a custom-built BD implementation? The key advantages of using the BD framework can be seen by reviewing Algorithms 1 and 2. The custom solving methods replace line 2 of Algorithm 1 and the custom cut generation methods replace lines 6 and 14 in Algorithm 2. The remainder of the solving process and the control mechanisms of the BD framework are left untouched. Thus, by using the BD framework the implementation effort of a custom BD algorithm is significantly reduced. In addition, and most importantly, many of the enhancement techniques, to be described later in this section, are still available for custom BD implementations.

Multiple decompositions A novel feature of the BD framework is the ability to provide multiple decompositions of the subproblem for the same original problem. This feature builds on the fact that given a master-subproblem split there are typically many different ways to decompose or reformulate the subproblem. Also, these decompositions may have properties that can be exploited algorithmically. For example, consider a subproblem for which there exists two formulations, one with a tighter relaxation bound than the other, but the former is more difficult to solve. It may be advantageous to initially solve

the quicker/weaker formulation to generate cuts before solving the slower/stronger formulation.

To inform the central control mechanism of the the preferred decomposition verification order, a priority is assigned to each BD plugin. This priority order is used when verifying BendersDecomp or BendersDecompLP, where the decompositions are called in decreasing order to execute Algorithm 2. Algorithm 2 is only called for a BD plugin if it has been called and returned Feasible for all BD plugins with a higher priority value. Only once all decompositions return Feasible, BendersDecomp or BendersDecompLP is verified as feasible.

4.2 Solving process

Improvements to the three-phase method Phase 1 of the three-phase method in the BD framework can be applied at every node in the branch-and-bound tree. However, this could result in too many executions of Algorithm 2 and, subsequently, the addition of a large number of BD cuts that could have a negative effect on the overall effectiveness of the BD algorithm. Various run time parameters have been provided to give more control over Phase 1 of the three-phase method in the BD framework, and alleviate some issues related to its use in a branch-and-cut solver.

The key run time parameters for the three-phase method are:

- `constraints/benderslp/maxdepth`: the depth in the branch-and-bound tree at which BendersDecompLP is verified when enforcing solutions from the LP relaxation—default 0 (i.e. only the root node). -1 means all nodes and `maxdepth` > 0 means all nodes up to a depth of `maxdepth`.
- `constraints/benderslp/depthfreq`: if `maxdepth` > 0 and `depthfreq` $= n > 0$, the BendersDecompLP is verified up to `maxdepth` and then for nodes at depths of $d = n, 2n, 3n, \ldots$. If `depthfreq` is set to 0, then BendersDecompLP is only verified at nodes up to a depth of `maxdepth`.
- `constraints/benderslp/stalllimit`: if `stalllimit` $= n > 0$, the verification of BendersDecompLP will be triggered for the next processed node if there has not been any improvement in the dual bound after processing the last n nodes.

Convex and CIP solving functions The two solving stages described in Section 2.3 is a commonly reported approach to employing BD when problem (2) comprises discrete variables [4, 34]. While common, there is a distinct lack of support for this algorithmic feature in general BD frameworks. As explained above, the design of the BD plugin provides the capabilities to employ the two stage solving process in all applications of the BD framework—from the most automated through to the most customised.

4.3 Heuristics

Primal heuristics have been used in many different contexts to improve the performance of the BD algorithm. The BD framework in SCIP has been extended with a general enhancement technique, the large neighbourhood Benders' search (LNBS), and a BD-motivated primal heuristic, the trust region

(TR) heuristic. For a detailed description of these two features the reader is referred to Maher [38]. Since the effect of employing the LNBS is evaluated in the computational experiments, a description of the important parameters is provided.

A number of run time parameters are provided to control the behaviour of the LNBS. The LNBS is activated by setting `<x>/lnscheck` to TRUE. The maximum depth at which the LNBS is employed is controlled by `<x>/lnsmaxdepth`, which is set to UNLIMITED by default. Below this depth, the LNS heuristics are executed as per default, i.e. finding improving solutions for the master problem without any verification of BENDERSDECOMP or BENDERSDECOMPLP. Finally, many BD cuts are generated while solving the LNS auxiliary problem by BD, which are valid for the master problem. These generated cuts can be transferred from the auxiliary problem to the master problem by setting `<x>/transfercuts` to TRUE.

4.4 Presolving

Initial auxiliary variable bounds The classical description of the BD algorithm initialises the master problem by omitting BD cuts. As a result, the LP relaxation of problem (3) becomes unbounded because φ is initially unrestricted with a non-zero objective coefficient. Following the verification of BENDERSDECOMPLP or BENDERSDECOMP, constraints will be added in the form of optimality cuts to provide a lower bound on φ. While it is not necessary to have an initial lower bound for φ, in some applications providing this bound can be beneficial computationally.

In the BD framework, there are two methods for providing a lower bound for φ. The first is an extension of the SMPS format. When the scenarios are specified using the `SCENARIOS` keyword, a user defined lower bound can be provided for each scenario. In the specification of the SMPS format, each scenario is identified in the `sto` file by headers similar to

```
SC SCEN1    ROOT    0.2    STAGE-2    0.0
```

In the original specification of the SMPS format [10], the `sto` file had only four entries for the scenario header. The modification to the SMPS format involves an additional entry in the scenario header (the final entry) to specify the lower bound, in this case 0.0. The lower bound LB is then used to form the constraint $\varphi \geq LB$ that is added to the master problem.

The second method for bounding the auxiliary variable is to compute a lower bound by solving the convex relaxation of problem (2) without fixing x, called the lower bounding problem (LBP). After solving the LP relaxation of LBP, LB is set to the dual bound and the constraint $\varphi \geq LB$ is added to the master problem.

LBP can be solved at two different parts of the SCIP solving process. A presolving step is included in the BENDERSDECOMP constraint handler. Specifically, the `conshdlrPresolBenders` callback function solves LBP for all subproblems and stores LB in the BD data structures. This presolving step is enabled by setting `constraints/benders/maxprerounds` to 1. Alternatively, by setting the parameter `<x>/updateauxvarbound` to TRUE, if the LP of the master problem is unbounded when first verifying BENDERSDECOMP (or BENDERSDECOMPLP), LBP is solved to compute a valid lower bound for φ.

Implied integer auxiliary variables A key feature of the BD framework is the capability to handle subproblems that contain discrete variables. If a BD subproblem is formulated as a CIP, it is possible that the objective function will always take integer values. For such subproblems, it can be advantageous to set the associated auxiliary variables as implied integer variables, as opposed to continuous. This can help in raising the dual bound and potentially prove optimality earlier than is possible when the auxiliary variable is a continuous variable.

4.5 Benders' decomposition cut generation

Cut strengthening Cut strengthening is a very important part of any state-of-the-art BD implementation. Since the first example of cut strengthening for BD by Magnanti and Wong [36], many methods have been proposed to improve the convergence through the selection of tighter BD cuts. Many of the proposed BD cut generation methods exploit the degeneracy of the subproblem dual solution [36, 44, 47]. Fundamental to these approaches is the use of a core point [18, 36, 47] or an analytical centre [19, 44].

The cut strengthening method implemented as part of the BD framework employs a simple in-out method, similar to that described by Fischetti et al. [18]. First an initial core point must be set. Six different options are provided for the initial core point $\hat{x}^o$: i) the first LP solution [20, 47], ii) the first primal feasible solution, iii) a solution vector of all zeros [42], iv) a solution vector of all ones [18,47] (both iii) and iv) could be infeasible; however Mercier et al. [42] and Papadakos [47] state that feasibility is not necessary for cut strengthening procedures), v) a relative interior point [36] and vi) the first feasible solution but reset with each new incumbent. Given an initial core point $\hat{x}^o$, Algorithm 1 is called from either BendersDecomp or BendersDecompLP using the separation point $\hat{x}_{SEP}$, which is given by

$$\hat{x}_{SEP} = \lambda\hat{x} + (1 - \lambda)\hat{x}^o.$$

If no improvement in the dual bound of the master problem is achieved after k calls to Algorithm 1, then the separation point is set to

$$\hat{x}_{SEP} = \lambda\hat{x} + \delta\mathbf{1},$$

where $|\delta| \ll 1$. Finally, if no dual bound improvement is observed after another k calls to Algorithm 1, then the separation point is set to $\hat{x}_{SEP} = \hat{x}$. If a dual improvement is observed at any stage, the cut strengthening process restarts, without reinitialising the core point. After each call to Algorithm 1, the core point is updated by

$$\hat{x}^o = \lambda\hat{x} + (1 - \lambda)\hat{x}^o.$$

Note that the cut strengthening procedure is only executed when verifying BendersDecomp and BendersDecompLP for the enforcement of the solution from the LP relaxation.

A set of run time parameters have been included to control the cut strengthening procedure. The values of λ, δ and k are set by the parameters `<x>/cutstrengthenmult`, `<x>/corepointperturb` and `<x>/noimprovelimit` respectively. The default values are 0.5, 10^{-6} and 5 respectively. The source of the initial core point is selected using the parameter `<x>/cutstrengthenintpoint`, and is set to v) by default.

Cutting while checking solution feasibility As shown in Figure 1, BENDERSDECOMP is also verified when checking the feasibility of candidate primal solutions; however, cuts need not be generated. While not always explicitly stated when BD implementations are described, it is possible to generate cuts during the checking of primal solutions.

One particular advantage of generating cuts from candidate solutions from primal heuristics are that they are more likely to be in the interior of the original problem feasible region than LP solutions. Similar to cut strengthening techniques that use core points, which are ideally in the interior of the original feasible region, cuts generated from candidate solutions are potentially stronger than those generated from LP solutions. A run time parameter (`<x>/cutcheck`) is provided to enable the generation of cuts when checking primal feasible solutions.

4.6 Handling numerical difficulties

Numerical difficulties can sometimes affect the generation of BD cuts. For example, numerical issues may cause a generated optimality cut to not eliminate the non-optimal solution $\hat{x}$, which could lead to the early termination of the BD algorithm. As such, mechanisms have been implemented in the BD framework to recover from this and related issues.

Solution polishing Consider a BD subproblem for which an optimality cut is generated; however, $\hat{\pi}^\top(g - B\hat{x}) \neq z(\hat{x})$, where $\hat{\pi}$ is the optimal dual solution of the subproblem. In this case, a valid cut can not be generated and added to the master problem. In the BD framework, an attempt to address this involve using the *solution polishing* capabilities within SOPLEX. Upon encountering a solution $(\hat{x}, \hat{\varphi})$ that is unable to generate a valid optimality cut, then solution polishing is enabled in SOPLEX and the LP is resolved. If the subproblem is dual degenerate, the perturbation caused by solution polishing may provide an alternative, hopefully more numerically stable, solution. If this alternative optimality cut is still not valid, then no optimality cut can be generated for the subproblem in the current iteration of the BD algorithm. In the event that solution polishing could not resolve the numerical difficulties, an alternative resolution method, such as subproblem merging (see following paragraph), is required.

Merging subproblem into master problem Consider a problem where there are n subproblems, with the index set given by $N := \{1, 2, \ldots, n\}$. The subproblem merging functionality has been implemented to address the situation when the subproblems indexed by $\bar{N} \subseteq N$ are unable to generate cuts—either due to the subproblems not being solved or valid cuts could not be generated—but a cut is required, i.e. the corresponding subproblems have been found to be not optimal. If the subproblems $N \backslash \bar{N}$ evaluate to FEASIBLE, then no cuts will be generated and the BD algorithm will terminate prematurely.

When subproblem merging is used, all subproblems indexed by $\bar{N}$ are classified as *merge candidates*. There are two types of merge candidates: i) priority candidates $\bar{N}_p$, the subproblems that must be merged into the master problem in order to resolve the verification issues (i.e, it has been identified that a subproblem will never solve without numerical issues or the required BD cuts are not available

for the problem type), and ii) default candidates $\bar{N}_d$, the subproblems that could be merged to resolve the current issues when verifying BENDERSDECOMP, and $\bar{N}_p \cup \bar{N}_d = \bar{N}, \bar{N}_p \cap \bar{N}_d = \emptyset$. If $\bar{N}_p \neq \emptyset$, then all subproblems i, where $i \in \bar{N}_p$, must be merged into the master problem in the current iteration. Otherwise, if $\bar{N}_p = \emptyset$ and $\bar{N}_d \neq \emptyset$ and no BD cuts are generated during the current verification, then at least one subproblem $i \in \bar{N}_d$ must be merged into the master problem to ensure the correctness of the BD algorithm.

The subproblem merging functionality is not available by default but can be implemented within custom BD plugins. The function `SCIPmergeBendersSubproblemIntoMaster` can be called to merge a subproblem identified by the parameter `probnumber` into the master problem. The subproblem merging is best performed within the `bendersPostsubsolveXyz` callback function, where the indices of the merge candidates are provided as a parameter.

The subproblem merging approach is utilised to address the numerical difficulties encountered when applying BD to general MIP instances based on structures detected by GCG.

4.7 Python interface

The Python interface PYSCIPOPT [37] has been extended with classes to enable the development of custom BD and BD cut plugins.

The different levels of modelling and solving flexibility with the BD framework, described in Section 4.1, feature in PYSCIPOPT. First, the capabilities to read instances in the SMPS format is available. To invoke the BD algorithm the appropriate parameter settings for the `sto` reader need to be specified. Second, helper functions have been provided to invoke the DEFAULT BD plugin. Specifically, `initBendersDefault` can be called with a single or dictionary of subproblem MODEL instances that are internally passed to the C-API function `SCIPcreateBendersDefault`. The call to `initBendersDefault` sets the appropriate settings to activate the BD constraint handlers—the three-phase method is activated by default. Third, custom BD cut generation methods and BD plugins are supported through the addition of the appropriate classes that act as wrappers for the respective C-API plugin types. Similar to the C-API, a custom BD cut generation method is included in the solution algorithm by calling `includeBenderscut`. Finally, a custom BD plugin can be specified and included by calling `includeBenders`. All of the fundamental and optional callback functions of the C-API are available within the Python `Benders` and `Benderscut` classes.

5 GCG – Automatic Benders' decomposition

The Generic Column Generation solver, GCG, extends SCIP to create a generic branch-price-and-cut solver. A major goal of the GCG project is to enable users to apply decomposition methods to general MIP problems without any knowledge of problem structure or expertise in the use of decomposition techniques. This goal is achieved through features such as constraint matrix structure detection, automatic decomposition for Dantzig-Wolfe reformulation, internal subproblem solving methods and column

generation-specific heuristics. In addition, GCG includes advanced column generation techniques, such as dual stabilisation, heuristic pricing and constraint-based branching rules.

The availability of the BD framework within SCIP provides the opportunity to employ the structure detection of GCG to develop an automated BD solver. To date, automated decomposition solvers have primarily focused on Dantzig-Wolfe reformulation and Lagrangian relaxation—making GCG one of the first solvers to automatically detect structure and apply BD to general MIP instances.

The development of an automatic BD solver involved two different extensions to GCG: providing an interface to the BD framework within SCIP and improving the detection scoring to identify suitable structures for BD. The first extension involved the development of a custom BD plugin, BendersGcg, that defined the fundamental callbacks of `bendersCreatesubGcg` and `bendersGetvarGcg` and the optional callback `bendersPostsubsolveGcg`. The BendersGcg plugin builds upon the data structures that are available within GCG. The subproblems are constructed during the initialisation process in the GCG core and are then stored within the BendersGcg data structures. The variable mappings are taken directly from the variable data that is generated during the automatic decomposition process. Within `bendersPostsolveGcg`, the subproblem merging feature (see Section 4.6) is employed to handle any case where the detected structure results in master and subproblems that are not compatible with the available cut generation methods, such as mixed integer master and subproblems. Further, if BendersDecomp is verified to True, the post solve callback constructs a solution for the original problem and stores it within the GCG core.

Prior to development of the BD framework, GCG only supported a single decomposition method, Dantzig-Wolfe reformulation. The development of the BD framework for SCIP offered an opportunity to support an additional decomposition method within GCG. The ability to select the preferred decomposition method is specified with the run time parameter `relaxing/gcg/mode`, where 0 is for Dantzig-Wolfe reformulation and 1 for BD. A final mode, option 2, instructs GCG not to perform any decomposition and directly solve the problem using SCIP[3].

The second extension to GCG involved modifications to the detection scoring methods to support the detection of structures suitable for the application of BD. The key features of this work, primarily undertaken by Bastubbe [6], are detailed in the release report for the SCIP Optimisation Suite [25]. From a practical point of view, the GCG distribution includes additional settings files to set the appropriate parameters for the detection of BD-suitable structures. These settings files are: `detect-benders`, `detect-benders-bin_master` and `detect-benders-cont_subpr`. The first detects structures without any further guidance from the user, i.e. no special structure with respect to the variable types is imposed. The second employs the best efforts to create a master problem that comprises only binary variables—a structure suitable for all default BD cut generation methods available in SCIP 6.0. Finally, the third settings file attempts to detect the classical structure to which BD is applied, mixed integer master problem and continuous subproblem.

[3]This final mode is a quirk arising from a technical detail encountered while developing the interface to the BD framework.

6 Computational Experiments

The computational experiments evaluate the performance of key features included in the BD framework. These experiments will highlight the effectiveness of classical and new enhancement techniques on a range of problem types. In particular, experiments will be performed with various parameter settings to evaluate the enhancement techniques of cut strengthening, cutting during feasibility checking, the LNBS and the three-phase method. While demonstrating the effectiveness of the BD framework, the computational experiments aim to provide insight into the most effective techniques for future users and developers of the BD algorithm. As a final comparison, the best performing settings for each enhancement technique will be compared against default SCIP, i.e. by not applying BD.

Instance test sets Six different problem types are used for the computational experiments.

- Cargo scheduling (CARGO): A stochastic scheduling and transportation problem. Originally presented by Mulvey and Ruszczyński [43] and collected from Felt [16]. The formulation for this problem can be found in Ariyawansa and Felt [5]. Consists of one core file and a series of stochastic files comprised of 2^n scenarios, where $n \in \{2, 3, \ldots, 11\}$. First stage is discrete and the second stage is continuous. 11 instances.

- Recoverable robust tail assignment problem (RRTAILASSIGN): Airline planning problem employing recoverable robustness. Core instances are based off the aircraft routing and SDAMRP instances using the F267_A49 schedule data from Maher et al. [39]. Instances with 5, 10, 20 and 30 scenarios are generated with objective uncertainty (modelling push-back recovery) and constraint coefficient uncertainty (modelling the removal of flight legs from routes). First stage is discrete and the second stage is continuous. 16 instances.

- Stochastic multiple knapsack problem (SMKP): A stochastic integer program with pure binary first and second stages. The instances 1 to 15, collected from the SIPLIB [2], are used for the experiments. 15 instances.

- Stochastic network interdiction problem (SNIP): A stochastic programming problem with discrete first stage and continuous second stage. The problem instances were original presented by Pan and Morton [46] and collected from Bodur et al. [11]. Instances 0 and 1 from classes 3 and 4 are used with budgets of 30, 40, 50, 60 and 70.

  20 instances.

In all experiments, 5 different random seeds were used to trigger performance variability and diversify the results. Each instance with a different random seed is treated as a separate instance.

The formulations for the CARGO and SNIP problems used in the computational experiments can be found in the associated references. For completeness, the formulations of RRTAILASSIGN and SMKP are presented in Appendix A.

To isolate the gain achieved from employing the selected enhancement technique, all experiments are evaluated against a branch-and-cut implementation of the BD algorithm without any enhancement techniques or features, labelled as *Basic*. While the parameter settings do not capture secondary effects that could result from the combination of different enhancement techniques, the results presented here are a guide to the most effective techniques for improving the BD algorithm. Finally, all instances are solved by default SCIP (i.e. no decomposition is performed), labelled as *Default*, to demonstrate the benefit of applying BD.

All computational experiments have been performed using SoPlex 4.0 as LP solver. The computational experiments have been conducted using a cluster consisting of Intel(R) Xeon(R) CPU E5-2670 2.5GHz processors with 64 GB RAM. A time limit of 3600 seconds is applied to all computational experiments.

Evaluation measure The instances selected to evaluate the BD framework vary greatly in terms of solving difficulty. Many of the instances are unable to be solved within the given time limit. Thus, the classical effectiveness measure of run time is not suitable for evaluating this diverse test set.

Gap integrals are measures capable of meaningfully evaluating instances that are unable to be solved when a time limit is imposed. The concept of gap integrals was originally proposed by Berthold [9], where the primal integral was used to evaluate the performance of heuristic algorithms. Briefly, given the gap (primal, dual or optimality) as a function of time, a gap integral is computed as the area under this curve in the range $t \in [0, T]$, where T is either the time limit or the time to optimality. In this paper the primal integral and dual integral are used to evaluate the effectiveness of the BD framework. A description of gap integrals is provided in Appendix B.

The vast majority of the enhancement techniques evaluated in these computational experiments are

	Cargo	RRTailAssign	Smkp	Snip
Basic	1,012.25	139,714.79	18,436.32	60,998.73
CS-First	988.44	7,670.05	18,403.46	33,353.55
CS-Inc	884.84	11,079.6	18,356.83	36,459.51
CS-LP	994.66	**7,253.22**	18,435.6	35,293.64
CS-RelInt	**700.77**	8,402.51	**17,766.0**	**29,803.45**
CutCheck	268.78	104,846.57	864.59	13,484.25
LNSCheck	1,008.57	139,220.9	20,552.96	60,360.91
LC-Depth20	**783.84**	139,268.35	18,384.12	60,685.06
TransferCuts	1,080.69	**135,641.04**	**5,874.42**	**40,001.79**
TC-Depth20	998.07	138,492.15	18,403.44	47,545.08
ThreePhase	**254.72**	**16,486.52**	**3,865.95**	**7,099.75**
TP-Depth10	349.01	17,236.49	4,150.51	11,564.82
TP-Depth5	263.05	16,743.57	4,029.78	7,458.01
Default	9,969.68	204,374.32	315.72	265,251.76

Table 1: Shifted geometric mean of the dual integral. Bold highlights the best setting for each settings group per instance set.

designed to improve the dual bound. Thus, the main performance benefit will be observed through the dual integral. For conciseness, only the results for the dual integral will be presented, except when discussing the LNBS where the results for both the primal and dual integrals will be presented. The results and related discussion for the primal integral is presented in Appendix C.

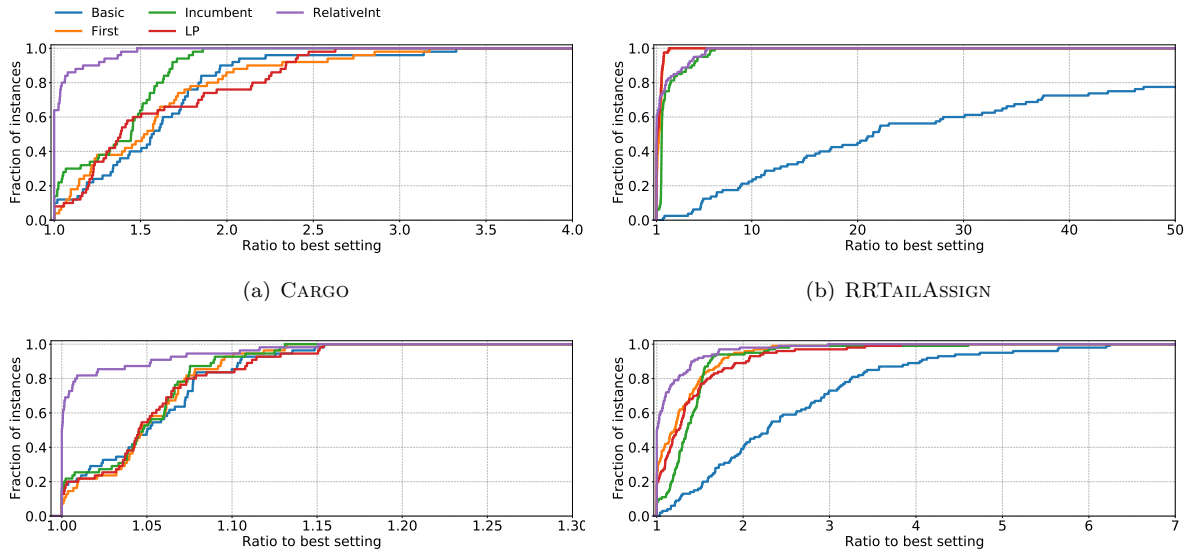
The results from the computational experiments are presented using performance profiles [15]. For conciseness, the largest ratio displayed on the performance profile is 50. It is deemed that comparing ratios greater than 50 does not contribute to the analysis of the results. Additionally, the shifted geometric mean (geomean) of the primal and dual integrals, with a shift of 1000, is presented for comparison. The shifted geometric means of the dual integral for all test settings is presented in Table 1. A complete set of instance-wise computational results is available from `https://www.stephenjmaher.com/publications`.

6.1 Cut strengthening

This experiment evaluates the effectiveness of initialisation points for the cut strengthening method. The initial core points and settings that will be evaluated are: the first LP solution (*CutStrengthen-LP*), first primal feasible solution (*CutStrengthen-First*), relative interior point (*CutStrengthen-RelativeInt*) and re-initialising the core point with every incumbent change (*CutStrengthen-Incumbent*).

Figure 2 presents performance profiles of the dual integral for *Basic*, *CutStrengthen-First*, *CutStrengthen-Incumbent*, *CutStrengthen-LP* and *CutStrengthen-RelativeInt*. The first observation from Figure 2 is that the cut strengthening technique does not always provide a benefit over *Basic*. Considering Cargo and Smkp, for all but *CutStrengthen-RelativeInt*, there appears to be little benefit from using cut strengthening. This is reinforced when looking at the geomean in Table 1; the settings *CutStrengthen-First*, *CutStrengthen-Incumbent* and *CutStrengthen-LP* all have little effect in improving the dual integral.

An important observation from Figure 2 is that *CutStrengthen-RelativeInt* appears to perform well on



(a) Cargo (b) RRTailAssign

(c) Smkp (d) Snip

Figure 2: Evaluating the cut strengthening techniques – Dual Integral.

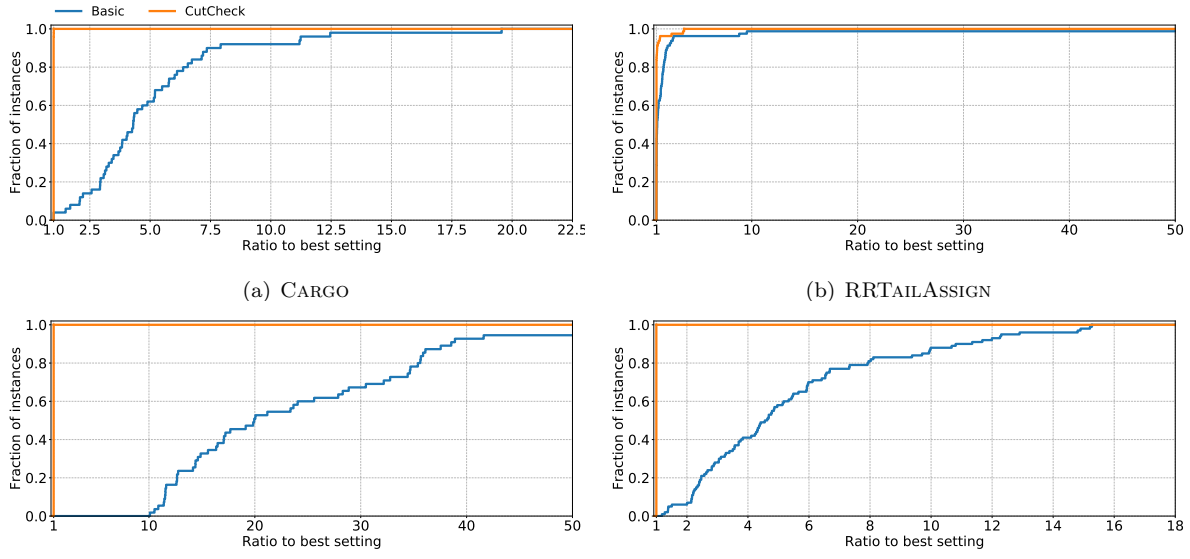
all test sets. While *CutStrengthen-RelativeInt* is not always the best performing setting for all test sets, it is routinely significantly better than *Basic*. For the CARGO, SMKP and SNIP test sets *CutStrengthen-RelativeInt* is the best performing setting with respect to the dual integral. The main exception is RRTAILASSIGN, where *CutStrengthen-First* and *CutStrengthen-LP* report a better performance, as observed with the geomean in Table 1. Overall, the relative interior point appears to be the best initialisation point for the cut strengthening techniques; however, the effectiveness of each initialisation point is still problem specific.

6.2 Cutting while checking solution feasibility

Generating cuts during the verification of solution feasibility, labelled as *CutCheck*, is a simple extension to the basic BD algorithm. Since the generation of BD cuts increases the size of the LP relaxation, there is a potential trade-off associated with the generation of cuts while verifying feasibility. This could be particularly important if there are a large number of solutions found throughout the branch-and-cut search tree.

The performance profiles for the dual integral when using *Basic* and *CutCheck* are presented in Figure 3. There is an obvious advantage to generating BD cuts while checking solution feasibility. In Figure 3, 4 out of the 6 test sets—CARGO, SMKP and SNIP—show that *CutCheck* is more effective than *Basic* on all considered instances. For the SMKP instances the improvement in dual integral by *CutCheck* over *Basic* ranges from 10.1× to 58.35×. These results show a great advantage to generating cuts during the feasibility check, especially when trying to quickly improve the dual bound.

Interestingly, the RRTAILASSIGN test set does not show as clear a signal as the above test sets. For the RRTAILASSIGN instances there is a dominance in the performance profile for *CutCheck* compared to *Basic*—suggesting the former setting is more effective on average than the latter—however there are still a number of instances where *Basic* reports a better performance than *CutCheck*. Comparing



(a) CARGO (b) RRTAILASSIGN

(c) SMKP (d) SNIP

Figure 3: Evaluating cutting on feasibility check – Dual Integral.

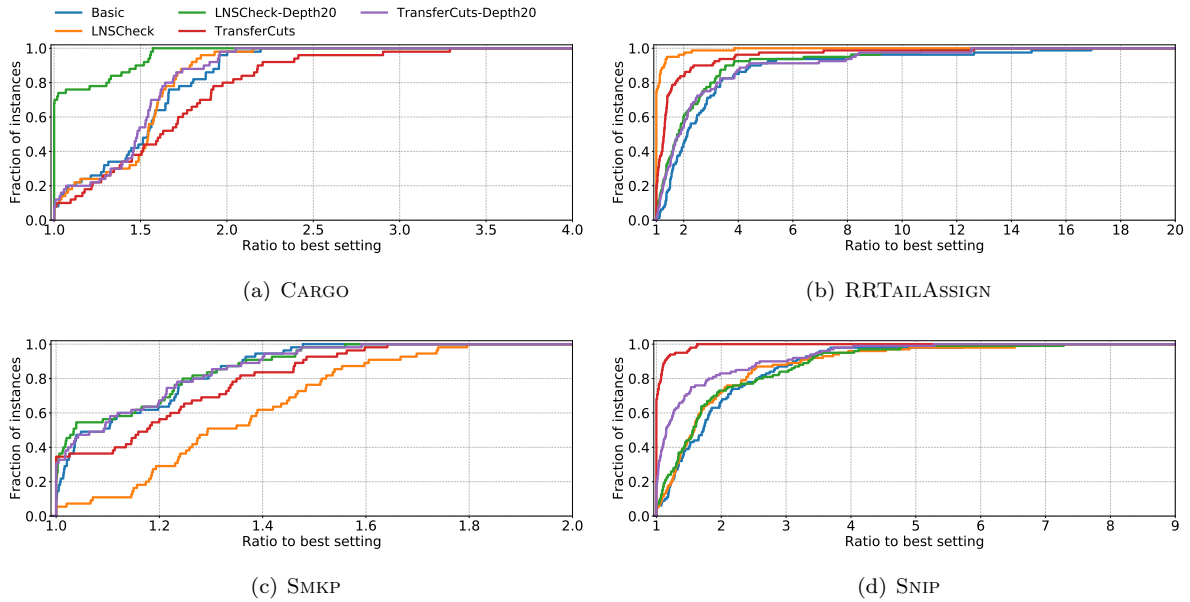
the geomean for *Basic* and *CutCheck* shows that the latter achieves a significantly better result. The RRTailAssign instances report 139,714.79 and 104,846.57 respectively, which is a 24.96% improvement.

6.3 Large neighbourhood Benders' search

The LNBS is a unique feature to the BD framework available within SCIP. This feature aims to improve the quality of the solutions found by LNS heuristics, but also with the transferring of cuts option it is possible to accelerate convergence through tighter dual bounds. The settings *LNSCheck* and *TransferCuts* perform the LNBS with and without transferring cuts from the LNS auxiliary problem to the master problem. Additionally, the settings *LNSCheck-Depth20* and *TransferCuts-Depth20* evaluate the effectiveness of the BD framework when performing the LNBS up to a depth of 20. Since the LNBS has the potential to improve both the primal and dual sides of the BD algorithm, both the primal and dual integrals will be discussed. Figures 4 and 5 present the performance profiles of the primal and dual integrals comparing *Basic* to the four settings that enable the LNBS. The geomean for the primal integral is presented in Table 2 and the dual integral is presented in Table 1.

The results for the primal integral, shown in Figure 4, are very mixed across the different test sets. An improvement in the performance of the BD algorithm can be observed for all LNBS settings for the RRTailAssign and Snip test sets. However, for the other test sets, some settings for the LNBS are seen to degrade performance, while for the Smkp instances the use of the LNBS appears to have very little positive effect.

Considering the RRTailAssign and Snip instances, Figure 4 and Table 2 show that there is a large benefit from employing the LNBS. For the RRTailAssign instances, *LNSCheck* achieves a significant reduction in the primal integral compared to all other settings. In particular, the use of *LNSCheck* reports a geomean of 38,973.0 compared to 85,833.83 for *Basic*—a 54.6% improvement in the primal integral. The primal integral results for the Snip instances suggest a benefit from transferring cuts, with



(a) Cargo (b) RRTailAssign

(c) Smkp (d) Snip

Figure 4: Evaluating the LNBS – Primal Integral.

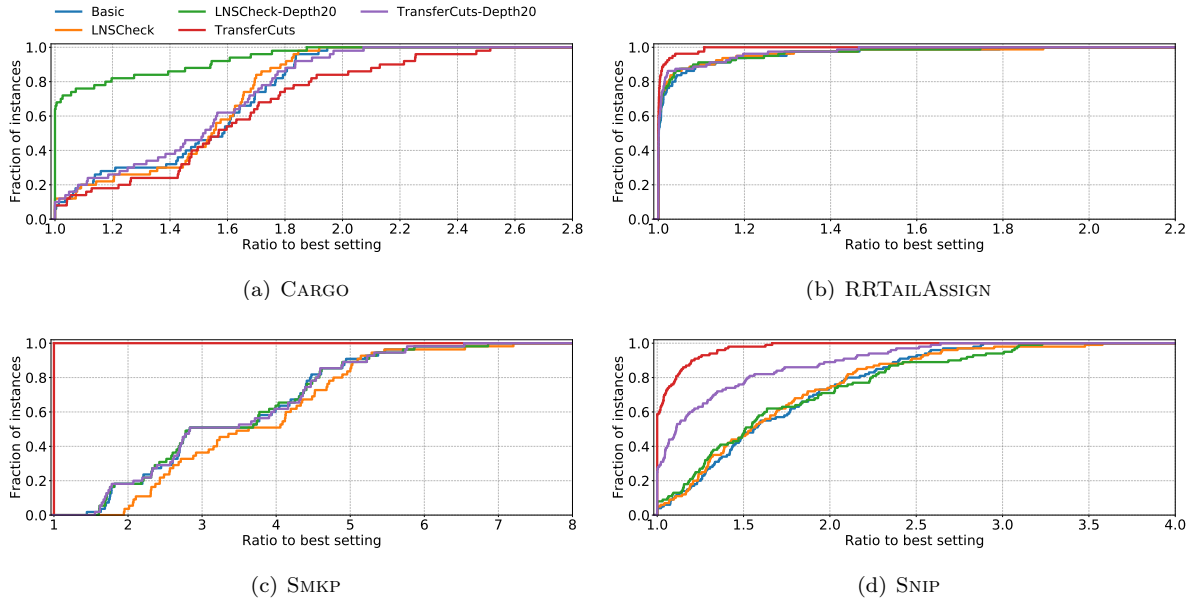
	CARGO	RRTAILASSIGN	SMKP	SNIP
Basic	412.39	85,833.83	19,459.09	14,675.89
LNSCheck	409.26	**38,973.0**	22,881.17	14,153.61
LC-Depth20	**321.58**	71,996.55	**19,310.67**	14,115.59
TransferCuts	450.65	51,831.37	20,373.68	**8,622.01**
TC-Depth20	404.11	76,142.82	19,393.1	11,458.79

Table 2: Shifted geometric mean of the primal integral when using the LNBS. Bold highlights the best performing setting per instance set.

TransferCuts and *TransferCuts-Depth20* achieving the largest reduction compared to *Basic*. In fact, *TransferCuts* and *TransferCuts-Depth20* achieve a 41.3% and 21.9% improvement in the primal integral over *Basic* respectively.

Vastly different results are observed for the dual integral compared to the primal integral. The most different result is seen for the SMKP test set where *TransferCuts* outperforms all other settings on all instances. This result also translates into improved solvability of the instances: 8 instance can be solved by *TransferCuts*, while all instances remain unsolved with the other settings.

The RRTAILASSIGN and SNIP instances also show an improvement in the dual integral when using some settings that employ the LNBS. For the RRTAILASSIGN test set, while the improvement in the dual integral using *TransferCuts* compared to *Basic* is only 2.91%, the 39.61% improvement in the primal integral makes this setting the best setting overall. Similar to the results for the primal integral, the SNIP test set reports a large improvement in the dual integral over *Basic* for the *TransferCuts* and *TransferCuts-Depth20* settings, an improvement of 34.4% and 22.1% respectively.



(a) CARGO (b) RRTAILASSIGN

(c) SMKP (d) SNIP

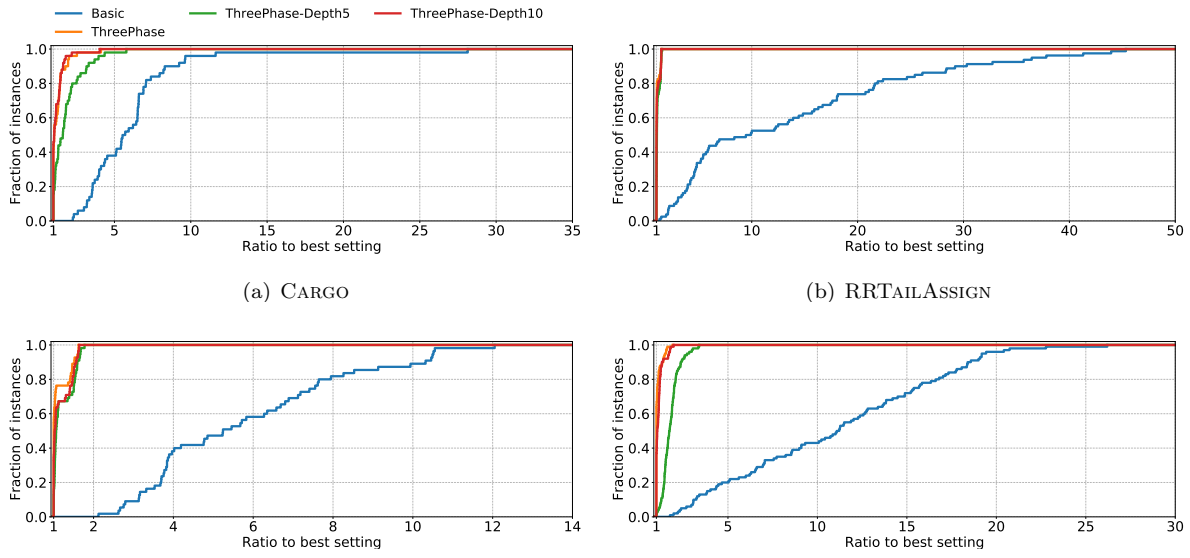
Figure 5: Evaluating the LNBS – Dual Integral.

6.4 Three-phase method

The three-phase method is a classical enhancement technique for the BD algorithm that can be employed throughout the branch-and-cut tree. In these experiments, the effect of the three-phase method will be evaluated when performed only at the root node (*ThreePhase*), up to a depth of 5 (*ThreePhase-Depth5*) and up to a depth of 10 (*ThreePhase-Depth10*).

The first observation from Figure 6 is that for any of the evaluated depth settings the three-phase method improves the effectiveness of the BD algorithm. A notable result comes from the RRTailAssign instances where an enormous improvement in the performance of the BD algorithm with the use of the three-phase method is observed. By employing the three-phase method many of the RRTailAssign instances can be solved at the root node.

Performing the three-phase method at nodes deeper than the root appears to have little performance advantage. While increasing the maximum depth to 5 deteriorates the performance with respect to the dual integral, Table 1 shows this degradation is not too large for many of the test sets. However, increasing the maximum depth to 10 greatly erodes the performance improvement achieved by using the three-phase method. Given that verifying BendersDecompLP is shown to improve the dual bound, an important area of research is to identify the nodes below the root where this will be most effective.



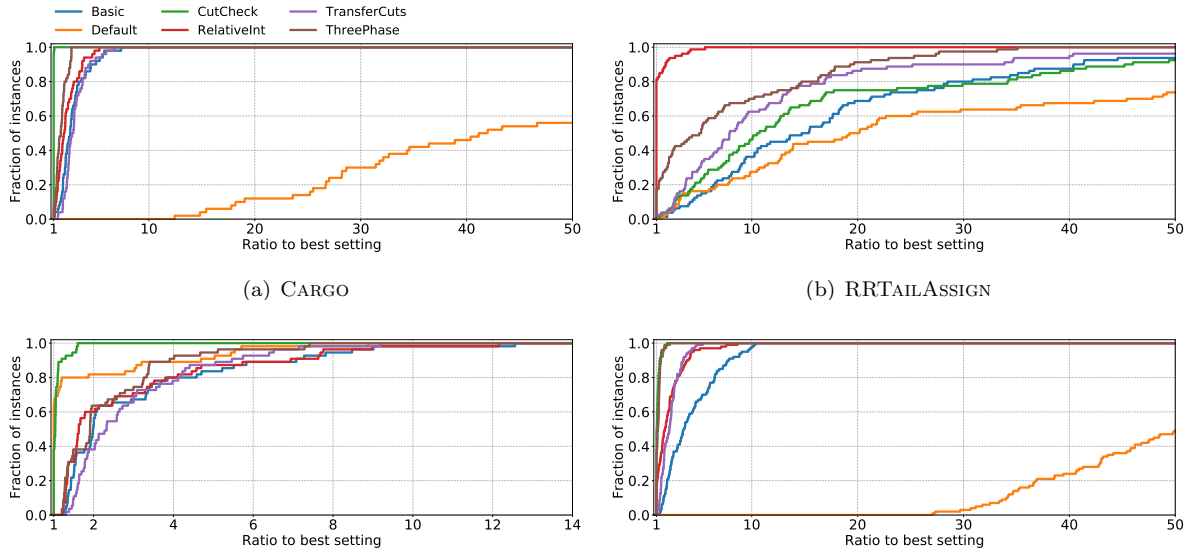
(a) Cargo (b) RRTailAssign

(c) Smkp (d) Snip

Figure 6: Evaluating the three-phase method – Dual Integral.

6.5 Comparison against default SCIP

A final evaluation of the BD framework assesses its effectiveness against solving the problem instances using SCIP without applying BD. From Sections 6.1–6.4, the best enhancement settings are: *CutStrengthen-RelativeInt*, *CutCheck*, *TransferCuts* and *ThreePhase*. These settings, including *Basic*, will be compared with respect to the primal integral and dual integral to evaluate the overall effectiveness of the BD framework and available enhancement techniques.



(a) Cargo (b) RRTailAssign

(c) Smkp (d) Snip

Figure 7: Evaluating *Default* against BD enhancements – Primal Integral.

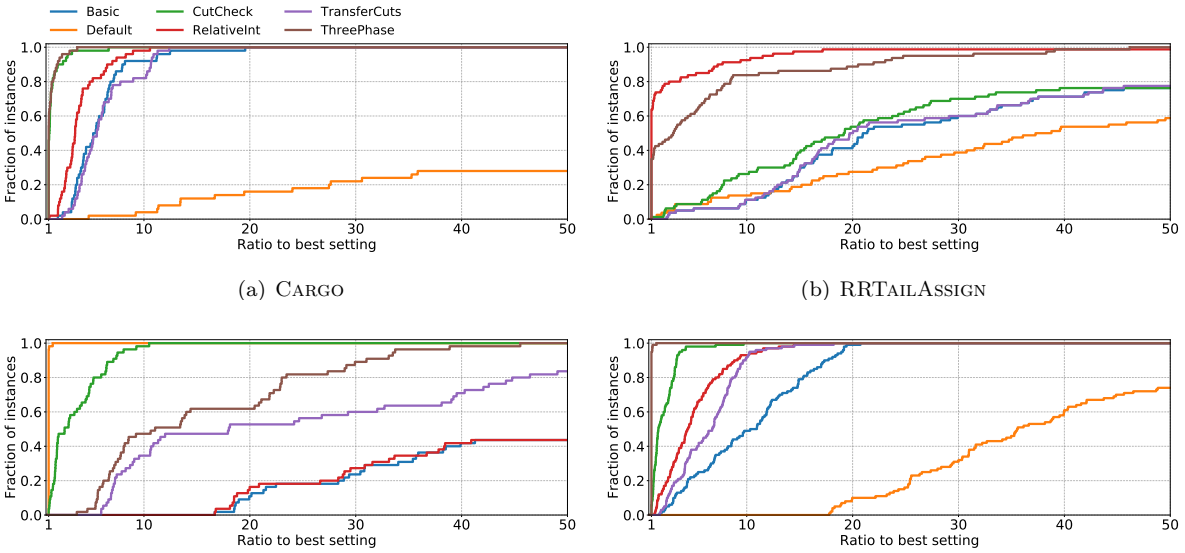
The first observation from the primal integral results presented in Figure 7 and Table 3 is that the application of BD achieves a significant improvement over *Default* for most instance classes. An improvement in the primal integral over *Default* by employing the basic BD implementation is observed for the Cargo, RRTailAssign and Snip—only the Smkp instances report a degradation in the primal integral from applying BD. This is to be expected, since the classes of instances have been selected because of their suitability for the application of BD; however, the magnitude of the improvement is greater than expected for many instances.

When considering the enhancement techniques, some instance classes report one to two orders of magnitude reduction in the primal integral. Specifically, *CutCheck* for the Cargo instances reduces the geomean from 6,459.23 to 170.09. Similar results can be observed for the RRTailAssign and Snip instances, where *CutStrengthen-RelativeInt* and *CutCheck*, respectively, achieve two orders of magnitude reduction in the primal integral. These results reinforce the value of enhancement techniques for improving the solvability of difficult problem instances.

The results presented in Figure 8 and Table 1 exhibit a similar reduction in the dual integral from the application of BD. For all instances where classical BD can be applied, i.e. mixed integer master problem

	Cargo	RRTailAssign	Smkp	Snip
Basic	412.39	85,833.83	19,459.09	14,675.89
CutCheck	**170.09**	70,059.37	**8,537.63**	**4,451.04**
CS-RelInt	335.53	**7,303.05**	17,786.1	7,621.07
TransferCuts	450.65	51,831.37	20,373.68	8,622.01
ThreePhase	265.44	28,323.26	16,478.77	4,675.18
Default	6,459.23	123,776.64	10,753.62	196,117.44

Table 3: Shifted geometric mean of the primal integral comparing *Default* against BD enhancements. Bold highlights the best performing setting per instance set.



(a) CARGO (b) RRTAILASSIGN

(c) SMKP (d) SNIP

Figure 8: Evaluating *Default* against BD enhancements – Dual Integral.

and continuous subproblem, the application of BD achieves a significant reduction in the dual integral. Interestingly, *Default* achieves the best dual integral across all enhancement technique setting for the stochastic programming problems with mixed integer subproblems, specifically the SMKP instances. This highlights that the available cuts for handling such problems are not very effective at improving the dual bound. An explanation for this is that the Laporte and Louveaux integer cuts provide a point-wise underestimation of the subproblem objective value, which are typically very weak cuts. A point of future work is to investigate BD cut generation methods to better handle stochastic integer programs.

In regards to the problem instances that BD is classically applied to, namely the CARGO, RRTAILASSIGN and SNIP instances, the BD framework is superior compared to *Default*. Similar to the results for the primal integral, the application of BD achieves one to two orders of magnitude improvement in the dual integral across this collection of instances. These results demonstrate the effectiveness of the BD framework at improving the solvability of decomposable instances when using SCIP.

7 Concluding remarks

A general Benders' decomposition framework has been developed for the solver SCIP. The plugin-based design and the numerous features provide numerous options for employing the Benders' decomposition algorithm. Through the GCG solver or by supplying an instance in the SMPS format, a user can employ Benders' decomposition without any knowledge of the underlying algorithm. The Benders' decomposition and Benders' cut plugins provide expert users the possibility to customise the algorithm with problem specific solution or cut generation methods. This flexibility provides users of SCIP the opportunity to employ the Benders' decomposition framework without the need to implement the full algorithm from scratch—only the most important components for research purposes need to be implemented.

An important aspect of this work is the availability of classical and novel enhancement techniques to support all uses of the general Benders' decomposition framework. A large collection of enhancement techniques have been implemented, such as cut enhancement techniques, the three-phase method and the LNBS, and are controlled through a range of run time parameters. All of these features are available when employing the Benders' decomposition framework in its most automated setting through to the most customised.

The computational experiments highlight some of the key features of the Benders' decomposition framework. These experiments focus on different enhancement techniques to provide some insight into their effectiveness on a broad range of test instances. The computational experiments show various results across the different settings and test instances. For example, the classical enhancement of cut strengthening is shown to have the best overall performance when using an initial core point that is in the relative interior of the master problem. Interestingly, cutting while checking the feasibility of solutions produces a significant improvement to the Benders' decomposition algorithm. A contribution of this work is the review of different enhancement techniques using the same computational and software platform.

The work presented in this paper is only the start of research into the branch-and-cut approach for the Benders' decomposition algorithm. As highlighted in Rahmaniani et al. [50], there are many open questions about how to best employ the various components of MIP and CIP solvers to improve the performance of the Benders' decomposition algorithm. In particular, primal heuristics, conflict analysis and propagation routines are valuable tools within MIP and CIP solvers that have received little attention from the Benders' decomposition research community. The Benders' decomposition framework in SCIP provides the perfect platform for conducting research into how to best utilise these algorithmic features for Benders' decomposition.

The Benders' decomposition framework in SCIP is provided as a tool for the wider academic community to support research into the Benders' decomposition algorithm. We hope to extend our understanding of the Benders' decomposition algorithm and help deliver advancements to this popular mathematical and constraint programming algorithm.

Acknowledgements

This work was support by the Engineering and Physical Sciences Research Council (EPSRC) grant EP/P003060/1. The author would like to thank Gregor Hendel, Ambros Gleixner, Leona Gottwald, Felipe Serrano, Stefan Vigerske and Jakob Witzig for performing code reviews for the general Benders' decomposition framework in SCIP 6.0 and SCIP 7.0. The author would also like to thank the anonymous reviewers and editors for the helpful comments that greatly improved the quality of this paper.

References

[1] T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.

[2] S. Ahmed, R. Garcia, N. Kong, L. Ntaimo, G. Parija, F. Qiu, and S. Sen. SIPLIB: a stochastic integer programming test problem library. See `http://www2.isye.gatech.edu/~sahmed/siplib/`. Last accessed: 21-04-2020.

[3] E. Álvarez Miranda, E. Fernández, and I. Ljubić. The recoverable robust facility location problem. *Transportation Research Part B: Methodological*, 79:93–120, 2015.

[4] G. Angulo, S. Ahmed, and S. S. Dey. Improving the integer L-shaped method. *INFORMS Journal on Computing*, 28(3):483–499, 2016.

[5] K. A. Ariyawansa and A. J. Felt. On a new collection of stochastic linear programming test problems. *INFORMS Journal on Computing*, 16(3):291–299, 2004.

[6] M. Bastubbe. Modular detection of model structure in integer programming. SCIP Workshop, RWTH Aachen, 2018.

[7] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4(1):238–252, 1962.

[8] M. Bergner, A. Caprara, A. Ceselli, F. Furini, M. Lübbecke, E. Malaguti, and E. Traversi. Automatic Dantzig-Wolfe reformulation of mixed integer programs. *Mathematical Programming*, 149(1–2):391–424, 2015.

[9] T. Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013.

[10] J. R. Birge, M. A. Dempster, H. I. Gassmann, E. Gunn, A. J. King, and S. W. Wallace. A standard input format for multiperiod stochastic linear programs. Technical Report WP-87-118, IIASA, Laxenburg, Austria, 1987.

[11] M. Bodur, S. Dash, O. Günlük, and J. Luedtke. Strengthened Benders cuts for stochastic integer programs with continuous recourse. *INFORMS Journal on Computing*, 29(1):77–91, 2017.

[12] Q. Botton, B. Fortz, L. Gouveia, and M. Poss. Benders decomposition for the hop-constrained survivable network design problem. *INFORMS Journal on Computing*, 25(1):13–26, 2013.

[13] G. Codato and M. Fischetti. Combinatorial benders' cuts for mixed-integer linear programming. *Operations Research*, 54(4):pp. 756–766, 2006.

[14] J.-F. Cordeau, G. Stojković, F. Soumis, and J. Desrosiers. Benders' decomposition for simultaneous aircraft routing and crew scheduling. *Transportation Science*, 35(4):375–388, 2001.

[15] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[16] A. Felt. Test-problem collection for stochastic linear programming. See `https://www4.uwsp.edu/math/afelt/slptestset.html`. Last accessed: 21-04-2020.

[17] FICO. FICO Xpress-Optimizer. See `https://www.fico.com/en/products/fico-xpress-optimization`. Last accessed: 21-04-2020.

[18] M. Fischetti, I. Ljubić, and M. Sinnl. Redesigning Benders decomposition for large-scale facility location. *Management Science*, 63(7):2146–2162, 2017.

[19] B. Fortz and M. Poss. An improved Benders decomposition applied to a multi-layer network design problem. *Operations Research Letters*, 37(5):359–364, 2009.

[20] G. Froyland, S. J. Maher, and C.-L. Wu. The recoverable robust tail assignment problem. *Transportation Science*, 48(3):351–372, 2014.

[21] M. Galati, T. Ralphs, and J. Wang. Computational experience with generic decomposition using the DIP framework. In *Proceedings of RAMP 2012*, 2012.

[22] G. Gamrath, T. Fischer, T. Gally, A. M. Gleixner, G. Hendel, T. Koch, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, F. Serrano, Y. Shinano, S. Vigerske, D. Weninger, M. Winkler, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 3.2. Technical Report 15-60, Zuse Institute Berlin, 2016.

[23] A. Geoffrion. Generalized benders decomposition. *Journal of Optimization Theory and Applications*, 10(4):237–260, 1972.

[24] A. M. Geoffrion and G. W. Graves. Multicommodity distribution system design by Benders' decomposition. *Management Science*, 20(5):822–844, 1974.

[25] A. Gleixner, M. Bastubbe, L. Eifler, T. Gally, G. Gamrath, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. E. Lübbecke, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, C. Schubert, F. Serrano, Y. Shinano, J. M. Viernickel, M. Walter, F. Wegscheider, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 6.0. Technical Report 18-26, Zuse Institute Berlin, 2018.

[26] Gurobi. Gurobi - the fastest solver. See `https://www.gurobi.com/`. Last accessed: 21-04-2020.

[27] W. E. Hart, J.-P. Watson, and D. L. Woodruff. Pyomo: modeling and solving mathematical programs in Python. *Mathematical Programming Computation*, 3(3):219, Aug 2011.

[28] C. Helmberg. The ConicBundle library for convex optimization, 2011.

[29] J. Huchette, M. Lubin, and C. Petra. Parallel algebraic modeling for stochastic optimization. In *Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages*, HPTCDL '14, pages 29–35, Piscataway, NJ, USA, 2014. IEEE Press.

[30] IBM. IBM ILOG CPLEX Optimization Studio. See `https://www.ibm.com/products/ilog-cplex-optimization-studio`. Last accessed: 21-04-2020.

[31] 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.

[32] K. Kim and V. M. Zavala. Algorithmic innovations and software for the dual decomposition method applied to stochastic mixed-integer programs. *Mathematical Programming Computation*, 10(2):225–266, 2018.

[33] L. Ladányi, T. K. Ralphs, and L. E. Trotter. Branch, cut, and price: Sequential and parallel. In M. Jünger and D. Naddef, editors, *Computational Combinatorial Optimization: Optimal or Provably Near-Optimal Solutions*, pages 223–260. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[34] G. Laporte and F. V. Louveaux. The integer L-shaped method for stochastic integer programs with complete recourse. *Operations Research Letters*, 13(3):133–142, 1993.

[35] M. Lubin, K. Martin, C. G. Petra, and B. Sandk. On parallelizing dual decomposition in stochastic integer programming. *Operations Research Letters*, 41(3):252 – 258, 2013.

[36] T. Magnanti and R. Wong. Accelerating Benders' decomposition: Algorithmic enhancement and model selection criteria. *Operations Research*, 29(3):464–484, 1981.

[37] S. Maher, M. Miltenberger, J. P. Pedroso, D. Rehfeldt, R. Schwarz, and F. Serrano. PySCIPOpt: Mathematical programming in Python with the SCIP Optimization Suite. In *Mathematical Software - ICMS 2016*, volume 9725, pages 301–307, 2016.

[38] S. J. Maher. Enhancing large neighbourhood search heuristics for benders' decomposition. Technical report, Lancaster University, 2019.

[39] S. J. Maher, G. Desaulniers, and F. Soumis. Recoverable robust single day aircraft maintenance routing problem. *Computers & Operations Research*, 51:130–145, 2014.

[40] A. Markert and R. Gollmer. Users guide to ddsip: A C package for the dual decomposition of two-stage stochastic programs with mixed-integer recourse. 2008.

[41] D. McDaniel and M. Devine. A modified Benders' partitioning algorithm for mixed integer programming. *Management Science*, 24(3):312–319, 1977.

[42] A. Mercier, J. Cordeau, and F. Soumis. A computational study of Benders' decomposition for the integrated aircraft routing and crew scheduling problem. *Computers & Operations Research*, 32(6):1451–1476, 2005.

[43] J. M. Mulvey and A. Ruszczyński. A new scenario decomposition method for large-scale stochastic optimization. *Operations Research*, 43(3):477–490, 1995.

[44] J. Naoum-Sawaya and S. Elhedhli. An interior-point Benders based branch-and-cut algorithm for mixed integer programs. *Annals of Operations Research*, 210(1):33–55, 2013.

[45] G. L. Nemhauser, M. W. Savelsbergh, and G. C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15(1):47–58, 1994.

[46] F. Pan and D. P. Morton. Minimizing a stochastic maximum-reliability path. *Networks*, 52(3):111–119, 2008.

[47] N. Papadakos. Practical enhancements to the Magnanti-Wong method. *Operations Research Letters*, 36(4):444–449, 2008.

[48] N. Papadakos. Integrated airline scheduling. *Computers & Operations Research*, 36(1):176–195, 2009.

[49] J. Puchinger, P. J. Stuckey, M. G. Wallace, and S. Brand. Dantzig-Wolfe decomposition and branch-and-price solving in G12. *Constraints*, 16(1):77–99, 2011.

[50] R. Rahmaniani, T. G. Crainic, M. Gendreau, and W. Rei. The Benders decomposition algorithm: a literature review. *European Journal of Operational Research*, 259(3):801–817, 2017.

[51] T. Ralphs and M. Güzelsoy. The SYMPHONY Callable Library for Mixed Integer Programming. In *Proceedings of the Ninth INFORMS Computing Society Conference*, pages 61–76, 2005.

[52] T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Parallel branch, cut, and price for large-scale discrete optimization. *Mathematical Programming*, 98(1-3):253–280, 2003.

[53] T. Santoso, S. Ahmed, M. Goetschalckx, and A. Shapiro. A stochastic programming approach for supply chain network design under uncertainty. *European Journal of Operational Research*, 167(1):96–115, 2005.

[54] E. Thorsteinsson. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. In *Principles and Practice of Constraint Programming - CP 2001*, pages 16–30. Springer, 2001.

[55] F. Vanderbeck. BaPCod—a generic branch-and-price code. See `https://raweb.inria.fr/rapportsactivite/RA2010/realopt/uid22.html`, 2005. Last accessed: 21-04-2020.

[56] J.-P. Watson, D. L. Woodruff, and W. E. Hart. PySP: modeling and solving stochastic programs in Python. *Mathematical Programming Computation*, 4(2):109–149, 2012.

[57] Y. Xu, T. K. Ralphs, L. Ladányi, and M. J. Saltzman. Computational experience with a software framework for parallel integer programming. *The INFORMS Journal on Computing*, 21(3):383–397, 2009.

[58] V. Zverovich, C. I. Fábián, E. F. D. Ellison, and G. Mitra. A computational study of a solver system for processing two-stage stochastic LPs with enhanced Benders decomposition. *Mathematical Programming Computation*, 4(3):211–238, 2012.

[59] SCIP: Solving Constraint Integer Programs. `http://scip.zib.de/`. Last accessed: 21-04-2020.

Supplementary Material

A Problem formulations

Six different problems are used for the computational experiments presented in Section 6. Each of the problem types have been taken from previously published papers: CARGO [5,16,43], RRTAILASSIGN [39], SMKP [4] and SNIP [11,46]. While the problems have been previously discussed, the formulations for the RRTAILASSIGN and SMKP problems have not been explicitly stated. The problem formulations used for the computational experiments for the RRTAILASSIGN and SMKP will be presented in this appendix.

A.1 Recoverable robust tail assignment problem

The RRTAILASSIGN problem is taken from the aircraft maintenance planning problem by Maher et al. [39]. This problem is a recoverable robust problem, with disruption scenarios denoted by the set S. The set of all aircraft is given by R and the set of all flights in the airline network is given by N. In Maher et al. [39], column generation is used to solve the planning and recovery aircraft maintenance routing problems. In the following formulation it is assumed that a subset of all possible columns have been generated for each aircraft $r \in R$, denoted as P^r and P^{sr} for scenario s, and that the optimal solution can be found from the columns in these subsets. The variables y_p^r (y_p^{sr}) equal 1 if aircraft r is assigned to flight route p (in scenario s). In the recovery problem, the variables z_i^s equal 1 if flight i is cancelled in scenario s. Finally, the variables ϵ_{sri}^+ and ϵ_{sri}^- are continuous variables to count the number of changes to the flights route for aircraft r in recovery scenario s compared to the planning stage.

The formulation for the RRTailAssign problem is given by

$$\text{minimise} \quad \sum_{r\in R}\sum_{p\in P^r} c_p^r y_p^r + \frac{1}{|S|}\sum_{s\in S}\left\{\sum_{r\in R}\sum_{p\in P_s^r} c_p^{sr} y_p^{sr} + \sum_{i\in N} d_i^s z_i + W\sum_{i\in N}(\epsilon_{sri}^+ + \epsilon_{sri}^-)\right\}, \tag{9}$$

$$\text{subject to} \quad \sum_{r\in R}\sum_{p\in P^r} a_{ip}^r y_p^r = 1 \qquad \forall i \in N, \tag{10}$$

$$\sum_{p\in P_s^r} y_p^r \le 1 \qquad \forall r \in R, \tag{11}$$

$$\sum_{r\in R}\sum_{p\in P_s^r} a_{ip}^{sr} y_p^{sr} + z_j^s = 1 \qquad \forall s \in S, \forall i \in N, \tag{12}$$

$$\sum_{p\in P_s^r} y_p^{sr} \le 1 \qquad \forall s \in S, \forall r \in R, \tag{13}$$

$$\sum_{p\in P^r} a_{ip}^r y_p^r - \sum_{p\in P^{sr}} a_{ip}^{sr} y_p^{sr} = \epsilon_{sri}^+ - \epsilon_{sri}^- \qquad \forall s \in S, \forall r \in R, \forall i \in N, \tag{14}$$

$$y_p^r \in \mathbb{Z}_+ \qquad \forall r \in R, \forall p \in P, \tag{15}$$

$$y_p^{sr} \in \mathbb{Z}_+ \qquad \forall s \in S, \forall r \in R, \forall p \in P, \tag{16}$$

$$z_i^s \in \{0,1\} \qquad \forall s \in S, \forall i \in N, \tag{17}$$

$$\epsilon_{sri}^+, \epsilon_{sri}^- \ge 0, \qquad \forall s \in S, \forall r \in R, \forall i \in N, \tag{18}$$

where the objective is to minimise the planning cost and the expected recovery costs. Additionally, the number of changes in the flight routes between the planning and recovery solution is penalised in the objective. Constraints (10) and (12) ensure that in planning and recovery each flight is assigned to exactly one aircraft, in the recovery case it could be cancelled. The parameters a_{ip}^r (a_{ip}^{sr}) equals 1 if flight i is included in flight route p for aircraft r (in scenario s). Every aircraft must be assigned at most one flight route, which is enforced with the constraints (11) and (13) respectively. Finally, constraints (14) count the number of changes between the planning and recovery flight routes for each aircraft. In the computational experiments, the integrality requirements in the second stage have been relaxed so that the classical BD algorithm can be employed.

A.2 Stochastic multiple knapsack problem

The Smkp is a two-stage stochastic programming problem with multiple binary knapsacks in both stages. The stochasticity in this problem is in the objective function coefficients for the second-stage. Three sets of items, I, J, K, can be packed into two sets of knapsacks, L, M. The parameters a_{il} and b_{jl} denote the weights of items i and j in knapsack l. Similarly, the parameters e_{im} and f_{km} denote the weights of

items i and k in knapsack m The formulation of the SMKP is given by

$$\text{minimise} \quad \sum_{i\in I} c_i x_i + \sum_{j\in J} d_j y_j + \frac{1}{|S|}\sum_{s\in S}\sum_{k\in K} g_k^s z_k^s \tag{19}$$

$$\text{subject to} \quad \sum_{i\in I} a_{il} x_i + \sum_{j\in J} b_{jl} y_j \geq \gamma_l \qquad \forall l \in L, \tag{20}$$

$$\sum_{i\in I} e_{im} x_i + \sum_{k\in K} f_{km} z_k^s \geq \delta_m \qquad \forall s \in S, \forall m \in M, \tag{21}$$

$$x_i \in \{0,1\} \qquad \forall i \in I, \tag{22}$$

$$y_j \in \{0,1\} \qquad \forall j \in J, \tag{23}$$

$$z_k \in \{0,1\} \qquad \forall k \in K. \tag{24}$$

B Gap integrals

Gap integrals are able to capture more information from the solution process than other classical evaluation measures. This is because the integral value is computed from the changes in the primal or dual bounds over time. As a result, gap integrals capture the trade-off between computational effort and the quality of the primal or dual bounds achieved.

Let $\delta_p(t)$ and $\delta_d(t)$ denote the primal and dual bounds at time t. The primal-dual gap at time t is computed by

$$\gamma(t) = \begin{cases} 0 & \text{if } |\delta_p(t)| = |\delta_d(t)| = 0, \\ \dfrac{\delta_p(t) - \delta_d(t)}{\max\{\delta_p(t), \delta_d(t)\}} & \text{if } 0 < \delta_p(t) \times \delta_d(t) < \infty, \\ 1 & \text{otherwise.} \end{cases} \tag{25}$$

The primal-dual integral is then given by

$$PDI = \int_0^T \gamma(t)\mathrm{d}t, \tag{26}$$

where T is either the run time to solve the instance or the time limit if the instance is unsolved.

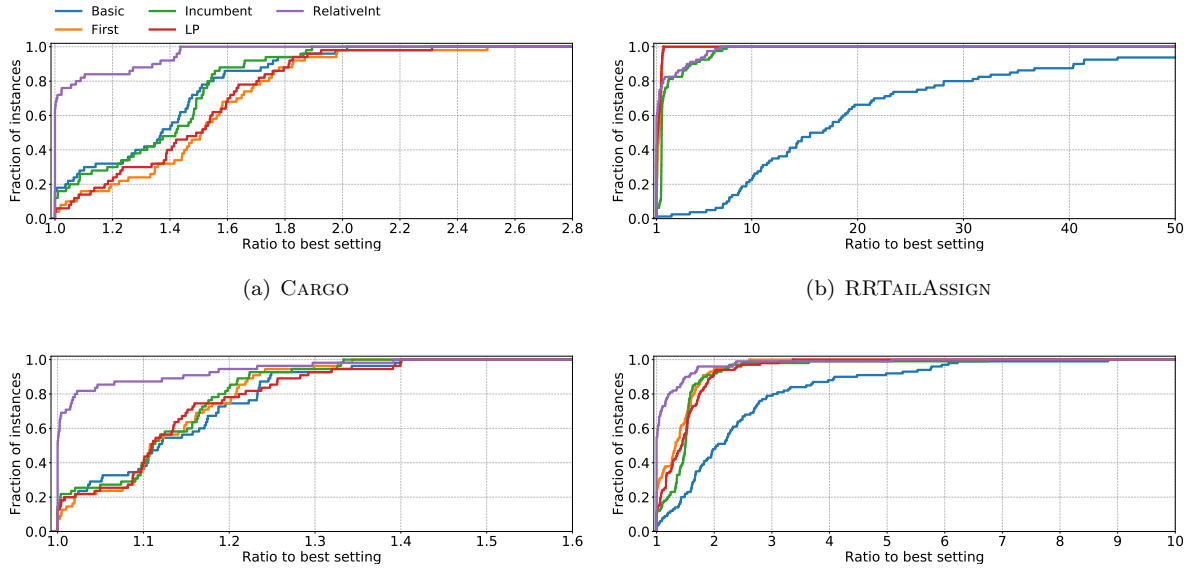
The primal integral is given by computing (26) after replacing $\delta_d(t)$ in (25) with the best known primal bound. Similarly, the dual integral is given by computing (26) after replacing $\delta_p(t)$ in (25) with the best known dual bound.

C Alternate integral results discussion

C.1 Cut strengthening – Primal integral

The performance profiles for the primal integral comparing *Basic* and various initialisation points for the cut strengthening methods are presented in Figure 9. Very similar results are observed for the primal integral compared to the dual integral. First, it appears that the use of cut strengthening does not always produce the most effective BD algorithm. The CARGO and SMKP test sets show that for all

settings except *CutStrengthen-RelativeInt* that there is little improvement over *Basic*. This is further reinforced with the geomean, presented in Table 4, where *CutStrengthen-LP*, *CutStrengthen-First* and *CutStrengthen-Incumbent* report a worse performance than *Basic* for the CARGO test set. For the other test sets there appears to be a benefit to using cut strengthening, regardless of the initialisation point.



(a) CARGO (b) RRTAILASSIGN

(c) SMKP (d) SNIP

Figure 9: Evaluating the cut strengthening techniques – Primal Integral.

	CARGO	RRTAILASSIGN	SMKP	SNIP
Basic	412.39	85,833.83	19,459.09	14,675.89
CS-First	448.56	6,651.57	19,365.27	8,994.42
CS-Inc	415.87	9,758.31	19,259.25	9,858.89
CS-LP	440.93	6,282.34	19,423.48	9,553.14
CS-RelInt	335.53	7,303.05	17,786.1	7,621.07

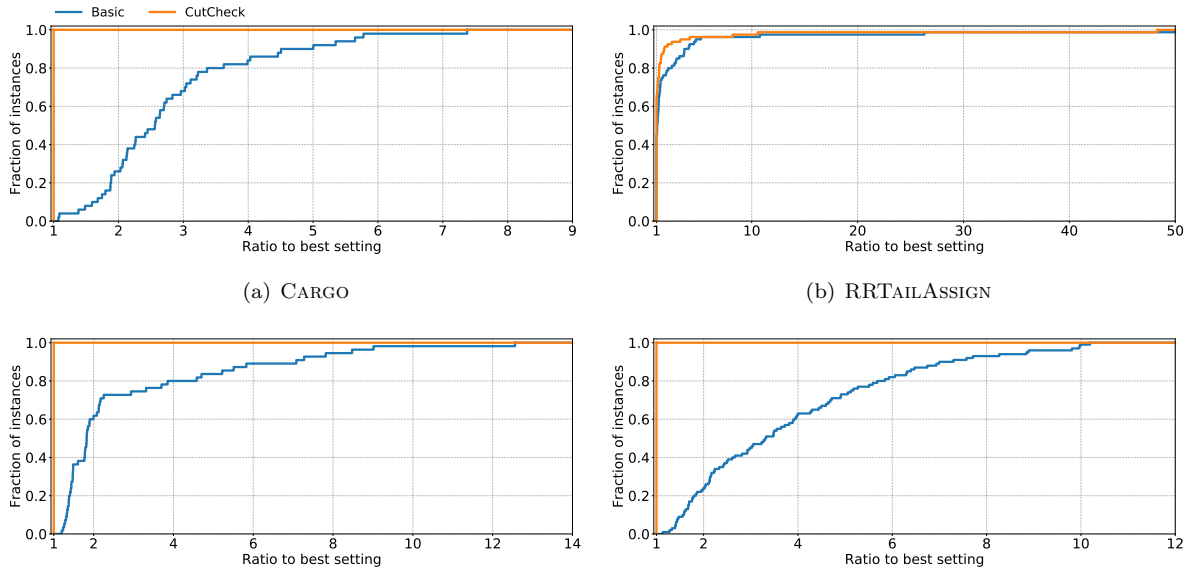
Table 4: Shifted geometric mean of the primal integral when using cut strengthening techniques.

Supporting the results for the dual integral, in Figure 9 and Table 4 the relative interior point appears to the best initialisation point for the cut strengthening methods. For all test sets, except RRTAILASSIGN, *CutStrengthen-RelativeInt* achieves the best geomean for the primal integral. In the case of the RRTAILASSIGN instances, the best performing setting is *CutStrengthen-LP*; however all settings achieve an improvement in the primal integral greater than 88%. Thus, while *CutStrengthen-RelativeInt* may not be the best performing setting, an improvement of 91.5% over *Basic* justifies its use.

C.2 Cutting while checking solution feasibility – Primal integral

The performance profiles for the primal integral are presented in Figure 10. Similar to the results for the dual integral in Section 6.2, there is a clear dominance of *CutCheck* over *Basic*. In fact, for the

CARGO, SMKP and SNIP the BD algorithm is more effective with *CutCheck* than *Basic* for all instances. A 12.56 times improvement in the primal integral by *CutCheck* over *Basic* is the largest single instance improvement observed for these three test sets. The RRTAILASSIGN instances tell a different story, with *Basic* being more effective that *CutCheck* on some instances; however, *CutCheck* is the most effective setting overall. Interestingly, the RRTAILASSIGN test set contains 3 instances where the primal integral for *Basic* is 10 to 100 worse than *CutCheck*.



(a) CARGO (b) RRTAILASSIGN

(c) SMKP (d) SNIP

Figure 10: Evaluating cutting on feasibility check – Primal Integral.

	CARGO	RRTAILASSIGN	SMKP	SNIP
Basic	412.39	85,833.83	19,459.09	14,675.89
CutCheck	170.09	70,059.37	8,537.63	4,451.04

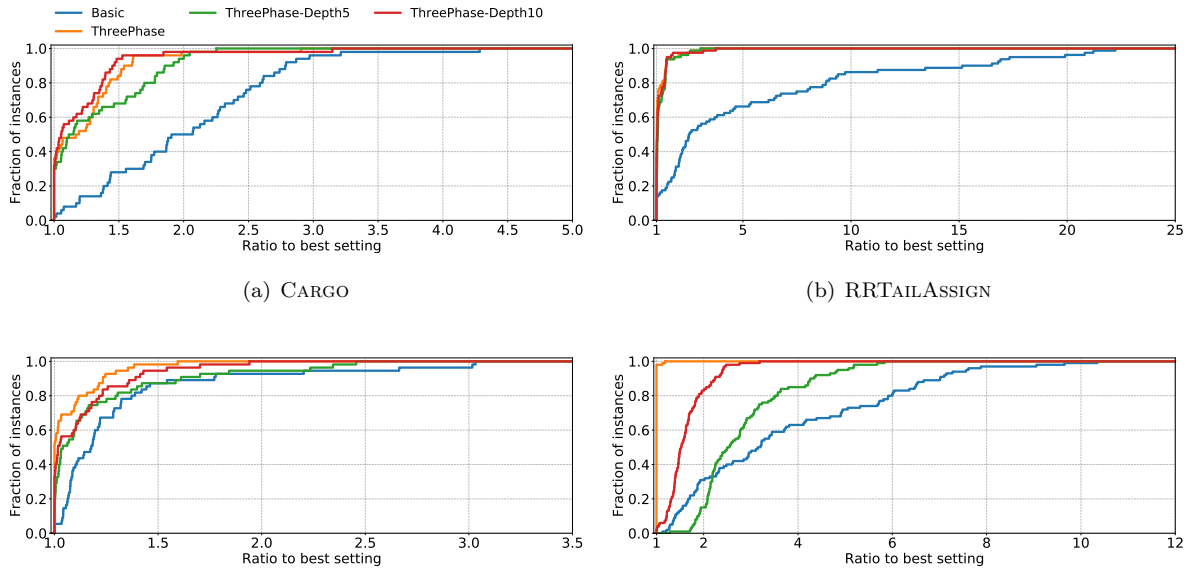
Table 5: Shifted geometric mean of the primal integral when cutting on feasibility check.

C.3 Three-phase method – Primal integral

The results for the primal integral are presented in Figure 11 and Table 6. Comparing the results for the dual integral from Section 6.4, a similar performance for the primal integral can be observed. In particular, there appears to be an overall benefit from using the three-phase method compared to *Basic*.

	CARGO	RRTAILASSIGN	SMKP	SNIP
Basic	412.39	85,833.83	19,459.09	14,675.89
ThreePhase	265.44	28,323.26	16,478.77	4,675.18
TP-Depth10	285.24	29,514.5	17,996.58	12,494.84
TP-Depth5	260.81	28,742.11	17,211.83	7,366.45

Table 6: Shifted geometric mean of the primal integral when using the three-phase method.



(a) CARGO (b) RRTAILASSIGN

(c) SMKP (d) SNIP

Figure 11: Evaluating the three-phase method – Primal Integral.

While the three-phase method is shown to improve the performance of the BD algorithm, in terms of the primal integral, the extent of the improvement is problem specific. For example, the RRTAILASSIGN instances exhibit a large improvement in the primal integral for most instances, up to 22 times that reported for *Basic*. Interestingly, for the SNIP test set there are no instances where *Basic* reports the best primal integral.