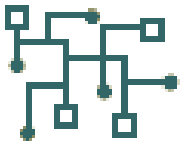

ISE

Industrial and Systems Engineering

Computational Experience with Generic Decomposition using the DIP Framework

Matthew V. Galati

Operations Research R & D, SAS Institute, Cary, NC 27513, US

Ted K. Ralphs and Jiadong Wang

Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, USA

COR@L Technical Report 12T-021

LEHIGH UNIVERSITY

COR@L

COMPUTATIONAL OPTIMIZATION RESEARCH AT LEHIGH

Computational Experience with Generic Decomposition using the DIP Framework

Matthew V. Galati*[1], Ted K. Ralphs†[2], and Jiadong Wang‡[2]

[1]Operations Research R & D, SAS Institute, Cary, NC 27513, US
[2]Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, USA

August 6, 2012

1 Introduction

Decomposition methods are techniques for exploiting the tractable substructures of an integer program in order to obtain improved solution techniques. In particular, the fundamental idea is to exploit our ability to either optimize over and/or separate from the convex hull of solutions to a given relaxation in order to derive improved methods of bounding the optimal solution value. These bounding methods can then be combined with associated methods for branching to obtain a complete branch-and-bound algorithm. This paper concerns a software package called DIP (*Decomposition in Integer Programming*) that simplifies the process of implementing decomposition-based bounding methods. Used in combination with the COIN-OR High Performance Parallel Search (CHiPPS) framework, DIP provides a complete decomposition-based MILP solver that can be used in either a generic mode in which no customization of algorithm components is required or as a framework in which users can replace generic implementations of various components with versions specialized to a particular problem class.

The philosophy of DIP is to remove as much implementational burden from the user as possible, while retaining the usual advantages of a decomposition-based methodology. The approach taken in DIP is substantially different than that taken by traditional "column-generation" frameworks, such as COIN/BCP [28], MINTO [24], ABACUS [17], and BapCod [35]. In such frameworks, implementing a bounding method based on Dantzig-Wolfe decomposition, for example, requires the user to formulate and work with the relaxation in an extended variable space. In addition, the user is typically required to provide implementations of customized methods for finding the decomposition and for branching, as well as a myriad of other possible auxiliary methods.

With DIP, any required reformulation is maintained internally and the user interacts with the framework only in terms of the original compact formulation. DIP also provides default implementations for all solver components and implements a wide range of decomposition-based algorithms,

*matthew.galati@sas.com
†ted@lehigh.edu
‡jiw508@lehigh.edu

including traditional methods based on Dantzig-Wolfe decomposition, Lagrangian relaxation, and cutting plane methods, as well as more advanced hybrid methods. As a generic solver, DIP is capable of automatically exploiting structure that is either indicated by the user through the use of the DipPy modeling language [25] or detected by automated analysis of the structure of the constraint matrix [36].

In several previous papers [27, 29, 30] and in the dissertation of Galati [11], we laid out the basic theoretical framework that forms the basis of DIP's implementation. In this paper, we first briefly review this framework (Section 3) and then discuss the basic design of DIP, with a focus on its use as a generic, decomposition-based MILP solver (Section 4). In Section 5, we provide some computational results using DIP in both its sequential and its multi-threaded parallel mode. Finally, in Section 6, we conclude. Before moving on, we briefly review previous work in Section 2 below.

2 Previous Work

Decomposition methods have been the subject of a rather large and varied literature. Algorithms for solving integer programs that involve some sort of decomposition can be found in every corner of the literature. Such methods are the preferred solution approaches for a wide range of important models arising in practice and have also been the basis for solution approaches for many well-known combinatorial problems. A small sample of the problems for which decomposition approaches have been proposed in the literature includes the Multicommodity Flow Problem [8], the Cutting Stock Problem [14, 33], the Traveling Salesman Problem [16], the Generalized Assignment Problem [15, 31], the Bin Packing Problem [34], the Axial Assignment Problem [2], The Steiner Tree Problem [20], the Single-Machine Scheduling Problem [32], the Graph Coloring Problem [22], and the Capacitated Vehicle Routing Problem (CVRP) [1, 7, 27, 9].

Developing usable software for implementing decomposition methods has been an often-pursued, but elusive goal. The most notable attempts at a framework for supporting techniques such as Dantzig-Wolfe decomposition are those that support the implementation of so-called "column-generation methods," which allow branch-and-bound to be performed with a formulation that has a large number of columns. Such formulations often arise from the application of Dantzig-Wolfe decomposition to an original "compact" formulation. The most well-developed and prominent frameworks supporting the implementation of column-generation-based algorithms are COIN/BCP [28], MINTO [24], ABACUS [17], and BapCod [35]. Very little work has been done on the so-called "generic decomposition methods" that are the focus of this paper. The only other work we know of in this direction is reported on in [4] and [12].

3 Overview of Decomposition

As we've already described, the goal of decomposition is to exploit tractable substructure. The most common approach to such exploitation is to relax a set of "complicating" constraints. This is the approach taken by the Dantzig-Wolfe, Lagrangian, and cutting plane methods. Substructure can also be exposed by fixing the values of a set of variables, i.e., considering restrictions of the original problem. This is the approach taken by Benders' decomposition. DIP supports decomposition based on relaxation of constraints. We review the principles of such techniques below.

3.1 Basic Principle

To simplify the exposition, we consider only pure integer linear programs (ILPs) with finite upper and lower bounds on all variables, so that the set of feasible solutions is finite. The framework can easily be extended to more general settings. For the remainder of the paper, we consider an ILP instance whose feasible set is the integer vectors contained in the polyhedron $\mathcal{Q} = \{x \in \mathbb{R}^n \mid Ax \geq b\}$, where $A \in \mathbb{Q}^{m \times n}$ is the constraint matrix and $b \in \mathbb{R}^m$ is the right-hand-side vector. Let $\mathcal{F} = \mathcal{Q} \cap \mathbb{Z}^n$ be the set of all *feasible solutions* to the ILP and let the polyhedron $\mathcal{P}$ be the convex hull of $\mathcal{F}$. In terms of this notation, the ILP is to determine

$$z_{IP} = \min_{x \in \mathcal{F}}\{c^\top x\} = \min_{x \in \mathcal{P}}\{c^\top x\} = \min_{x \in \mathbb{Z}^n}\{c^\top x \mid Ax \geq b\}, \tag{IP}$$

where $c \in \mathbb{R}^n$. By convention, $z_{IP} = \infty$ if $\mathcal{F} = \emptyset$ (the problem is infeasible).

To apply the principle of constraint decomposition, we consider the relaxation of (IP) defined by

$$\min_{x \in \mathcal{F}'}\{c^\top x\} = \min_{x \in \mathcal{P}'}\{c^\top x\} = \min_{x \in \mathbb{Z}^n}\{c^\top x \mid A'x \geq b'\}, \tag{R}$$

where $\mathcal{F} \subset \mathcal{F}' = \{x \in \mathbb{Z}^n \mid A'x \geq b'\}$ for some $A' \in \mathbb{Q}^{m' \times n}, b' \in \mathbb{Q}^{m'}$ and $\mathcal{P}'$ is the convex hull of $\mathcal{F}'$. As usual, we assume that there exist practical[1] algorithms for optimizing over and/or separating from $\mathcal{P}'$. With respect to $\mathcal{F}'$, let $[A'', b''] \in \mathbb{Q}^{m'' \times n''}$ be a set of additional constraints needed to describe $\mathcal{F}$, i.e., $[A'', b'']$ is such that $\mathcal{F} = \{x \in \mathbb{Z}^n \mid A'x \geq b', A''x \geq b''\}$. We denote by $\mathcal{Q}'$ the polyhedron described by the inequalities $[A', b']$ and by $\mathcal{Q}''$ the polyhedron described by the inequalities $[A'', b'']$. Hence, the initial LP relaxation is the linear program defined by $\mathcal{Q} = \mathcal{Q}' \cap \mathcal{Q}''$, and the *LP bound* is given by

$$z_{LP} = \min_{x \in \mathcal{Q}}\{c^\top x\} = \min_{x \in \mathbb{R}^n}\{c^\top x \mid A'x \geq b', A''x \geq b''\}. \tag{LP}$$

Note that $[A', b']$ and $[A'', b'']$ are often a partition of the rows of $[A, b]$ into a set of "nice constraints" and a set of "complicating constraints", but this is not a strict requirement.

Optimization and/or separation over $\mathcal{P}'$ may be more tractable than over $\mathcal{P}$ for a number of reasons. First, it may simply be that the resulting problem has a known structure that can be analyzed combinatorially or for which we have well-developed solution techniques. This first case arises frequently when the original problem is formed by adding side constraints to a well-studied base problem. Second, the resulting subproblem may itself decompose naturally due to identifiable bordered block-diagonal structure of the matrix A' (see Section (3.3)). In this case, optimization over $\mathcal{P}'$ can be accomplished by optimizing over each of these blocks independently (possibly in parallel). This technique is particularly effective when the blocks resulting from the decomposition are identical, since this indicates some degree of symmetry in the original model.

3.2 Decomposition Methods

Once identified, the decomposition can be exploited using three general classes of method. All three methods compute what we refer to generically as the *decomposition bound*, which is the quantity

$$z_D = \max_{x \in \mathcal{P}' \cap \mathcal{Q}''} c^\top x. \tag{D}$$

[1]The term *practical* is not defined rigorously, but denotes an algorithm with a "reasonable" average-case running time.

It is evident that $z_{LP} \leq z_D \leq z_{IP}$ and the goal is to choose a decomposition for which the inequality on the left is strict, so that we get an improvement in the bound obtained by solving the LP relaxation.

The **Dantzig-Wolfe method** (DWM) [5] computes this bound by dynamically constructing a (partial) inner approximation of $\mathcal{P}'$ in order to solve the so-called *master problem*, which is to compute

$$z_{DW} = \min_{\lambda \in \mathbb{R}_+^{\mathcal{F}'}} \{c^\top (\sum_{s \in \mathcal{F}'} s\lambda_s) \mid A''(\sum_{s \in \mathcal{F}'} s\lambda_s) \geq b'', \sum_{s \in \mathcal{F}'} \lambda_s = 1\}. \tag{DWLP}$$

This is accomplished by iteratively generating members of $\mathcal{F}'$ (usually extreme points of $\mathcal{P}'$), which then form the columns of (DWLP). In each iteration, a column with negative reduced cost is generated. The problem of generating such a column is an optimization problem over $\mathcal{P}'$ that we refer to as the *column generation subproblem* in this setting. The method terminates when no column with negative reduced cost can be found. The solution to (DWLP) can be interpreted as weights by which members of $\mathcal{F}'$ can be combined to yield an optimal solution

$$\hat{x}_{DW} = \sum_{s \in \mathcal{F}'} s\hat{\lambda}_s, \tag{MAP}$$

to (D).

The **Lagrangian method** (LM) can be thought of as a method for solving the dual of (DWLP). For a given vector of *dual multipliers* $u \in \mathbb{R}_+^{m''}$, the *Lagrangian relaxation* [6, 3, 23, 19] of (IP) is given by

$$z_{LR}(u) = \min_{s \in \mathcal{F}'} \{(c^\top - u^\top A'')s + u^\top b''\}. \tag{LR}$$

It is then easily shown that $z_{LR}(u)$ is a lower bound on z_{IP}. The problem

$$z_{LD} = \max_{u \in \mathbb{R}_+^{m''}} \{z_{LR}(u)\} \tag{LD}$$

of maximizing this bound over all choices of dual multipliers is a dual to (IP) called the *Lagrangian dual* (LD) and also provides a lower bound z_{LD}, which we call the *LD bound*. A vector of multipliers $\hat{u}$ that yield the largest bound are called *optimal (dual) multipliers*. The Lagrangian function $z_{LR}(u)$ is piecewise linear concave in u and its maximum can hence be found efficiently by any number of methods for maximization of concave functions.

In contrast to the DWM and LM, the **cutting plane method** (CPM) relies on the construction of a (partial) *outer* approximation of $\mathcal{P}'$ to compute the *CP bound*, which is

$$z_{CP} = \min_{x \in \mathcal{P}'} \{c^\top x \mid A''x \geq b''\}. \tag{CPLP}$$

Note that $\hat{x}_{DW}$, as defined in (MAP), is an optimal solution to this augmented linear program. We refer to the augmented linear program above as the *cutting plane LP* (CPLP). It can be solved in an iterative fashion by solving a *separation problem* over $\mathcal{P}'$ in each iteration. It is well known that the separation problem for a polyhedron is polynomially equivalent to the optimization problem over the same polyhedron.

Geoffrion showed equality of these three bounds with the decomposition bound.

Theorem 1 *(Geoffrion [13])* $z_{IP} \geq c^\top \hat{x}_{DW} = z_{LD} = z_{DW} = z_{CP} = \min\{c^\top x \mid \mathcal{P}' \cap \mathcal{Q}''\} \geq z_{LP}$.

These three algorithms can be abstracted into a common algorithmic framework. In all three cases, the bounding problem is an iterative procedure consisting of an alternation between solution of an *restricted master problem*, solved to obtain solution information, and a *subproblem*, solved to obtain additional information about the polyhedral structure of $\mathcal{P}'$ that is then used to augment the master problem. This iteration is continued until an appropriate termination criterion is reached.

We illustrate the principles discussed above with two examples. Example 1 illustrates the application of constraint decomposition to a simple two-variable ILP and Figure 1 demonstrates computation of the bound graphically. Example 2 describes a well-known combinatorial problem called the *Generalized Assignment Problem* that exhibits block structure.

Example 1: SILP Consider the following ILP in two variables:

$$
\begin{array}{lrll}
\min & x_1 & & \\
\text{s.t.} & 7\,x_1 - x_2 \geq & 13, & (1) \\
& x_2 \geq & 1, & (2) \\
& -x_1 + x_2 \geq & -3, & (3) \\
& -4\,x_1 - x_2 \geq & -27, & (4) \\
& -x_2 \geq & -5, & (5) \\
& 0.2\,x_1 - x_2 \geq & -4, & (6)
\end{array}
\qquad
\begin{array}{lrll}
& -x_1 - x_2 \geq & -8, & (7) \\
& -0.4\,x_1 + x_2 \geq & 0.3, & (8) \\
& x_1 + x_2 \geq & 4.5, & (9) \\
& 3\,x_1 + x_2 \geq & 9.5, & (10) \\
& 0.25\,x_1 - x_2 \geq & -3, & (11) \\
& x \in & \mathbb{Z}^2. & (12)
\end{array}
$$

In this example, we let

$$
\begin{aligned}
\mathcal{P} &= \operatorname{conv}\left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1)-(12)\right\}, \\
\mathcal{Q}' &= \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (1)-(6)\right\}, \\
\mathcal{Q}'' &= \left\{x \in \mathbb{R}^2 \mid x \text{ satisfies } (7)-(11)\right\}, \text{ and} \\
\mathcal{P}' &= \operatorname{conv}(\mathcal{Q}' \cap \mathbb{Z}^2).
\end{aligned}
$$

In Figure 1(a), we show the associated polyhedra, where the set of feasible solutions $\mathcal{F} = \mathcal{Q}' \cap \mathcal{Q}'' \cap \mathbb{Z}^2 = \mathcal{P}' \cap \mathcal{Q}'' \cap \mathbb{Z}^2$ and $\mathcal{P} = \operatorname{conv}(\mathcal{F})$. Figure 1(b) depicts the continuous approximation $\mathcal{Q}' \cap \mathcal{Q}''$, while Figure 1(c) shows the improved approximation $\mathcal{P}' \cap \mathcal{Q}''$. For the objective function in this example, optimization over $\mathcal{P}' \cap \mathcal{Q}''$ leads to an improvement over the LP bound obtained by optimization over $\mathcal{Q}$. ■

Example 2: GAP The Generalized Assignment Problem (GAP) is that of finding a minimum cost assignment of n tasks to m machines such that each task is assigned to precisely one machine subject to capacity restrictions on the machines. With each possible assignment, we associate a binary variable x_{ij}, which, if set to one, indicates that machine i is assigned to task j. For ease of notation, let us define two index sets $M = \{1, \ldots, m\}$ and $N = \{1, \ldots, n\}$. Then an ILP

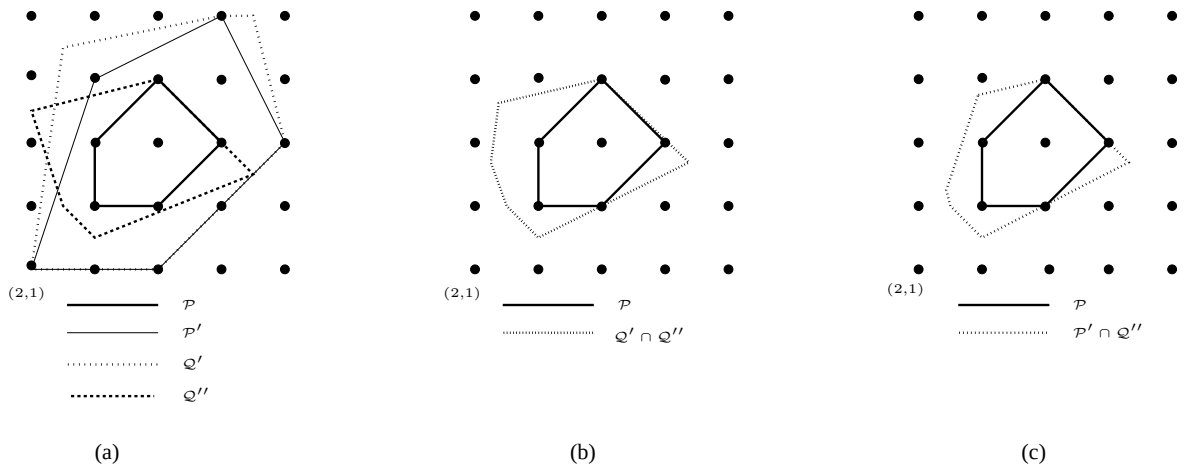

Figure 1: Polyhedra (Example 1: SILP)

formulation of GAP is as follows:

$$\min \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} \ ,$$

$$\sum_{j \in N} w_{ij} x_{ij} \ \leq \ b_i \ \forall i \in M, \tag{13}$$

$$\sum_{i \in M} x_{ij} \ = \ 1 \ \ \forall j \in N, \tag{14}$$

$$x_{ij} \in \{0,1\} \qquad \forall i, j \in M \times N. \tag{15}$$

In this formulation, equations (14) ensure that each task is assigned to exactly one machine. Inequalities (13) ensure that for each machine, the capacity restrictions are met.

There are several possible decompositions that result in tractable subproblems, but we focus here on relaxation of the capacity constraints, which can be seen as "linking constraints" involving all machines. Relaxing these constraints, we get the following:

$$\mathcal{Q}' \ = \left\{ x_{ij} \in \mathbb{R}^+ \ \forall i, j \in M \times N | \ x \text{ satisfies (13)} \right\} \text{ and}$$
$$\mathcal{Q}'' = \left\{ x_{ij} \in \mathbb{R}^+ \ \forall i, j \in M \times N | \ x \text{ satisfies (14)} \right\}.$$

The relaxation is a set of knapsack problems. As we will discuss later, this is an example of a problem with block structure—the optimization over $\mathcal{P}'$ reduces to a set of independent knapsack problems, one for each machine, which can be solved effectively in practice. ■

3.3 Generic Decomposition

Aside from the detailed implementational difficulties typically associated with decomposition-based methods, even the conceptual design of a decomposition-based algorithm for a specific problem has

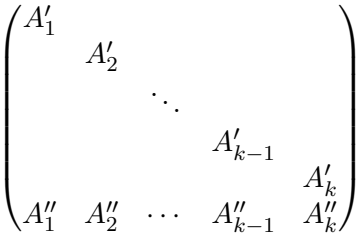

Figure 2: Singly-bordered block-diagonal matrices

traditionally required custom methods for

- identifying the decomposition (which constraints to relax);
- formulating and solving the subproblem (either optimization or separation over $\mathcal{P}'$);
- formulating and solving the master problem; and
- performing the branching operation.

It is, however, possible to develop fully generic methodologies for these components of the overall algorithm and thereby obtain a fully generic decomposition-based algorithm for solving an MILP. One of the goals of DIP is to be a platform for testing this idea. Here, we focus on some of the parts of DIP's implementation that provide this capability.

Identifying the Decomposition. One of the key motivators of the original Dantzig-Wolfe technique was the potential block structure of the constraint matrix. In Section 4, we describe how DIP identifies block structure either automatically or with the help of the user. Here, we focus on defining what bordered block-diagonal structure is and why its identification results in candidates for decompositions.

The process of identifying bordered block structure can be thought of as that of identifying a permutation of the rows and columns of A that exhibits the structure shown in Figure 2. The matrix A' is comprised of k disjoint blocks of nonzero elements. When A' has this structure, the column generation subproblem decomposes naturally into k independent (and smaller) MILPs. This is the property we wish to exploit and the motivation for identifying this structure.

The nonzeros of A'' can appear in any column and the corresponding constraints serve to link the k disjoint sets of constraints of $[A', b']$. These are generally referred to as the *coupling constraints*. The structure shown in Figure 2, is called a *singly-bordered* block-diagonal matrix. It is also possible to have a set of columns that are allowed to have nonzeros in any row, in which case we have a *doubly-bordered* structure. We restrict the discussion here to the singly-bordered case. For details on the doubly-bordered case in this setting, see [4].

To illustrate, let us return to our second example.

Example 2: GAP (continued) It should be clear that when choosing the capacity constraints (13) as the relaxation, we are left with a set of independent knapsack problems, one for each machine in M. Therefore, we can define the feasible regions for each *block* as follows:

$$\mathcal{P}'_k = \text{conv } \{x_{ij} \in \mathbb{R}^+ \; \forall i, j \in M \times N | \; x \text{ satisfies (13) and (15)}, x_{ij} = 0 \; \forall i \in M \setminus \{k\}, j \in N\},$$
$$\mathcal{Q}'' = \quad \{x_{ij} \in \mathbb{R}^+ \; \forall i, j \in M \times N | \; x \text{ satisfies (14)}\}.$$

For clarity, let us consider an example with $n = 3$ tasks and $m = 2$ machines. The formulation looks as follows:

$$\begin{aligned}
\min \; & x_{11} + x_{12} + x_{13} + x_{21} + x_{22} + x_{23}, & & \\
& x_{11} + x_{12} + x_{13} & \leq b_1, & (16)\\
& x_{21} + x_{22} + x_{23} & \leq b_2, & (17)\\
& x_{11} + x_{21} & = 1, & (18)\\
& x_{12} + x_{22} & = 1, & (19)\\
& x_{13} + x_{23} & = 1, & (20)\\
& x_{11}, \; x_{12}, \; x_{13}, \; x_{21}, \; x_{22}, \; x_{23} & \in \{0, 1\}. &
\end{aligned}$$

The polyhedron $\mathcal{Q}''$ is formed from constraints (18)-(20), while the polyhedron $\mathcal{P}'$ is formed from (16)-(17). From the structure of the matrix, it is clear that the two capacity constraints are separable, forming two independent blocks. ■

Example 3: Generic MILP In this example, we illustrate by using the instance *a1c1s1* from MIPLIB 2003 how block structure can be identified in DIP when the model is an (initially) unstructured, generic MILP that has been read from a file in a text-based format. Since no existing standard format for specifying MILP instances allows for the notation of block structure, such structure must either be identified by the user in an auxiliary "block file" or automatically by DIP in a separate pre-processing step. In our example, the structure of the matrix as initially read from the file (i.e., the position of the nonzeros in the constraint matrix) is shown on the left in Figure 3. After permuting the rows and columns of the matrix, the hidden block structure is evident. The same matrix after permutation is show on the right in Figure 3. This structure was identified by hypergraph partitioning procedures described in [36] and passed to DIP with the instance.

Branching Methods. In the cutting plane method, the branching operation is most commonly performed by choosing an integer-constrained variable with index i for which $\hat{x}_i \notin \mathbb{Z}$, where $\hat{x} \in \mathbb{R}^n$ is the solution to (LP). We then produce two disjoint subproblems by enforcing $x_i \leq \lfloor \hat{x}_i \rfloor$ in one subproblem and $x_i \geq \lceil \hat{x}_i \rceil$ in the other. The branching constraints are enforced by simply updating the variable bounds in each subproblem.

To accomplish this same division into subproblems using (DWLP) as the bounding problem, we have two approaches. One approach is to let all variable bounds be considered explicitly as constraints in the compact formulation by moving them into the constraints $[A'', b'']$. Using the

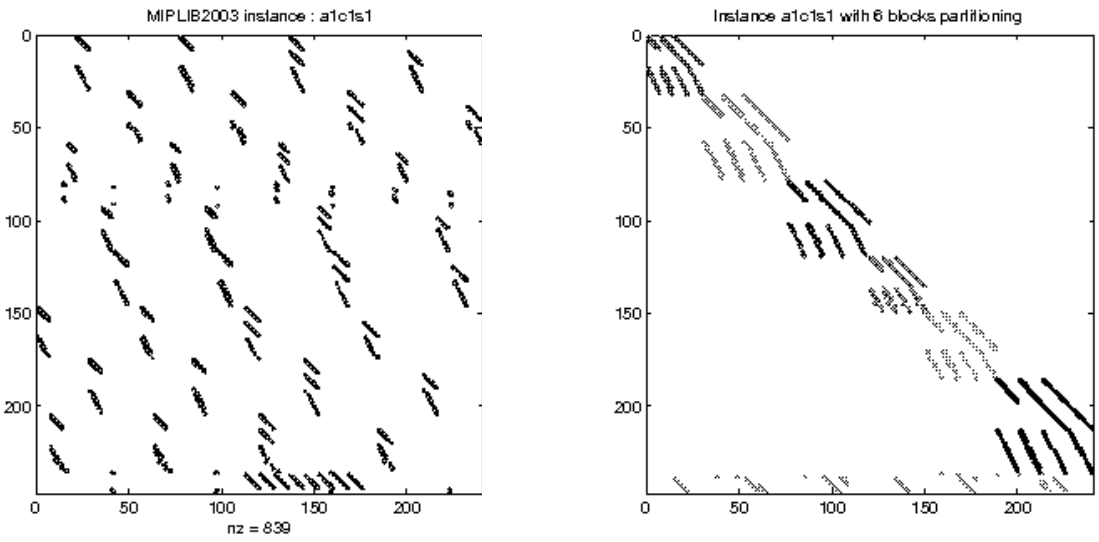

Figure 3: MIPLIB2003 instance *a1c1s1* with original structure and detected block structure

mapping (MAP), the variable bounds can then be written as general constraints of the form $l_i \leq \left(\sum_{s\in\mathcal{E}} s\lambda_s\right)_i \leq u_i, \forall i \in I$ in the extended formulation. As in standard branch-and-cut, after solving (DWLP), we choose an integer variable with index i whose value $\left(\sum_{s\in\mathcal{E}} s\hat{\lambda}_s\right)_i = \hat{x}_i \notin \mathbb{Z}$ in the compact space is fractional and produce two disjoint subproblems by enforcing $\left(\sum_{s\in\mathcal{E}} s\lambda_s\right)_i \leq \lfloor \hat{x}_i \rfloor$ in one subproblem and $\left(\sum_{s\lambda_s\in\mathcal{E}} s\lambda_s\right)_i \geq \lceil \hat{x}_i \rceil$ in the other. Since these are *branching constraints* rather than the standard *branching variables*, we enforce them by adding them directly to $[A'', b'']$.

A second approach is to enforce the branching decisions in the subproblem itself by directly adjusting the variable bounds. In this case, we must also be sure to remove all columns of the master problem associated with members of $\mathcal{P}'$ that were previously generated which would violate the imposed branching constraints.

Solving the Subproblem. Although one of the traditional strengths of decomposition methods is that they allow the use of specialized algorithms for solving the subproblem, this is not actually a requirement. Effective implementations can be obtained by using a generic MILP solver. This is particularly true when the decomposition is achieved by exploiting block structure, since this can lead to a substantial reduction in the size of the MILPs that need to be solved as compared to that of the original compact formulation.

4 The DIP Framework

The theoretical and algorithmic framework described in Section 3 lends itself nicely to implementation within a generic software framework. In this section, we review some features of the open-source C++ framework DIP (Decomposition for Integer Programming). DIP is a flexible framework that provides extensible implementations of the decomposition algorithms in Section 3 and much more. Advanced users of DIP have the ability to override almost any algorithmic component, enabling

Project Name	Description
ALPS	The Abstract Library for Parallel Search, an abstract base layer for implementations of various tree search algorithms
CBC	COIN-OR branch-and-cut, an LP-based branch-and-cut solver for MILP
CGL	Cut Generator Library, a library of cutting-plane generators
CLP	COIN-OR LP, a linear programming solver that includes simplex and interior point solvers
CoinUtils	Utilities, data structures, and linear algebra methods for COIN-OR projects
OSI	Open Solver Interface, a uniform API for calling embedded linear and mixed integer programming solvers

Table 1: COIN-OR Projects used by DIP

the development of custom variants of the aforementioned methods.

DIP has been released as part of the COIN-OR repository [10]. It is built on top of several existing COIN-OR projects and relies on others for its default implementations. Table 1 contains a list of projects that are used by DIP. One of those projects, ALPS (the Abstract Library for Parallel Search), which provides overall management of the branch-and-bound tree, is the search-handling layer of the COIN-OR High-Performance Parallel Search (CHiPPS) framework [37].

DIP treats inner methods, such as Dantzig-Wolfe decomposition and Lagrangian relaxation, in the same framework as classical cutting-plane methods. This allows for seamless integration of these decomposition methods into a wide range of hybrid and advanced algorithms. The mapping (MAP) between the compact formulation and the extended formulation, which was discussed in Section 3, is key to this design. This allows the framework to simultaneously manage and integrate the dynamic generation of cuts and variables. In addition, this also greatly simplifies the branching operation, as discussed in Section 3.3. These two facts allow for a simplified user interface in which all user interaction with the framework is in terms of the compact formulation, including the specification of methods of generating valid inequalities in CPM, solving the subproblem in LM or DWM, branching, generating heuristic solutions, and other customizations.

4.1 User Interface

To use DIP in its most basic mode as a generic solver, the main requirement is that the user provides an explicit description of the polyhedron $\mathcal{Q}''$ and either an implicit description of $\mathcal{P}'$ (through an optimization or separation oracle) or an explicit description of the polyhedron $\mathcal{Q}'$, as discussed in Section 3. This can be done either by

- building the models in memory using the C++ interface;
- by reading the model from a file in one of several standard text-based formats and specifying the decomposition in a separate text file; or
- by building the model in the modeling language interface provided by DipPy.

We provide details of each of these modes below and illustrate with examples in Section 4.2.

C++ Interface. Fundamentally, DIP is a C++ library that uses inheritance to enable the ability to perform customizations. The user interface is divided into two separate classes, an *applications interface*, encapsulated in the class `DecompApp`, in which the user may provide implementations of application-specific methods (e.g., solvers for the subproblems), and an *algorithms interface*, encapsulated in the class `DecompAlgo`, in which the user can modify DIP's internal algorithms, if desired. The `DecompAlgo` class and its derivatives provide implementations of all of the methods described in Section 3.

The base class `DecompApp` provides an interface for the user to define the model and the application-specific algorithmic components to define the model. Through methods in this class, the user may specify the model either by directly building parts of it in memory (using the `DecompConstraintSet` class) or by specifying oracles for optimization and/or separation over $\mathcal{P}'$.

Once the model has been defined, the user has the choice of overriding a number of different methods that will affect the algorithmic processing. Here, we point out some of the most commonly used methods. For a full description, reference the doxygen API documentation provided at [10].

`DecompApp::solveRelaxed()`: With this method, the user can provide an algorithm for solving the subproblem. In the case of an explicit definition of $\mathcal{Q}'$ as an MILP, the user does not have to override this method—DIP simply uses the built-in MILP solver (CBC) or another OSI-supported MILP solver that has been linked in the setup process.

`DecompApp::generateCuts()`: With this method, the user can provide an algorithm for solving the separation subproblem. DIP already has generators for most of the common classes of generic valid inequalities provided by the CGL built in. For many applications, the user will have specific knowledge of methods for generating valid inequalities that can greatly enhance performance.

`DecompApp::isUserFeasible()`: This function is used to determine whether or not a given solution, $\hat{x}$, is feasible to the original model. In the cases in which the user gives an explicit definition of $\mathcal{Q}''$, this function is unnecessary, since all constraints are explicit and feasibility can be checked using an automated routine. However, when a full description of $\mathcal{Q}''$ is not given a priori, a function for checking feasibility is necessary to determine whether a given solution is feasible.

The methods pertaining to the base algorithm are encapsulated in `DecompAlgo`. These methods would not typically be modified by the user, but can be if desired. The main loop is agnostic to the particular bounding method being employed and alternates between solving a master problem to obtain solution information and solving a subproblem to generate new polyhedral information, as described in Section 3.

Python Interface. The modeling language DipPy, developed at the University of Auckland and described in [25], is an algebraic modeling language built as an extension of the Python programming language. It provides a convenient and intuitive interface to DIP, allowing the user to easily specify the model and the decomposition in a natural way. Examples of the uses of DipPy are provided in Section 4.2.

```
1  void GAP_DecompApp::createModels(){
2
3     //get information about this problem instance
4     int           nTasks     = m_instance.getNTasks();     //n
5     int           nMachines  = m_instance.getNMachines(); //m
6     const int *   profit     = m_instance.getProfit();
7     int           nCols      = nTasks * nMachines;
8
9     //construct the objective function
10    m_objective = new double[nCols];
11    for(i = 0; i < nCols; i++) { m_objective[i] = profit[i]; }
12    setModelObjective(m_objective);
13
14    DecompConstraintSet * modelCore = new DecompConstraintSet();
15    createModelPartAP(modelCore);
16    setModelCore(modelCore);
17
18    for(i = 0; i < nMachines; i++){
19       DecompConstraintSet * modelRelax = new DecompConstraintSet();
20       createModelPartKP(modelRelax, i);
21       setModelRelax(modelRelax, i);
22    }
23 }
```

Listing 1: DIP createModels for GAP example

Command Line Interface. Models may also be provided to DIP through a command line interface by reading an explicit description of the model in one of several common text-based formats. A separate block file may be used to specify the block structure of the matrix. If no such decomposition is provided, then an auxiliary subroutine for finding the block structure automatically is invoked.

4.2 Examples

To better understand how a user might interact with the framework, this section provides code snippets that show how to solve the GAP using DIP and DipPy through C++ and Python, respectively.

GAP in C++. Recall that when choosing to relax the capacity constraints (13), we are left with a set of independent knapsack problems, one for each machine in M. In Listing 1, we show how one defines these blocks when setting the model components[2]. At lines 14–16, we create and set the description of $\mathcal{Q}''$, the assignment constraints, using the method `setModelCore`. Then, at lines 18–22, we create and set each block using the `setModelRelax` method. Notice that the third argument of this method allows the user to specify the block identifier. For each block defined, DIP knows how to handle the algorithmic accounting necessary for the associated reformulations.

Since each block defines an independent binary knapsack problem, we want to take advantage

[2]For the sake of conserving space, in each of the listings in this chapter, we remove all of the standard error checking, exception handling, logging, and memory deallocation. Therefore, these listings should be considered *code snippets*. For a complete set of source code, please see [10].

```
1  DecompSolverStatus GAP_DecompApp::solveRelaxed(const int       whichBlock,
2                                                 const double  * costCoeff,
3                                                 DecompVarList & newVars){
4
5     DecompSolverStatus status = DecompSolStatNoSolution;
6     if(!m_appParam.UsePisinger) { return status; }
7
8     vector<int>      solInd;
9     vector<double>   solEls;
10    double           varCost    = 0.0;
11    const double   * costCoeffB = costCoeff + getOffsetI(whichBlock);
12
13    status = m_knap[whichBlock]->solve(whichBlock, costCoeffB,
14                                       solInd, solEls, varCost);
15    DecompVar * var = new DecompVar(solInd, solEls, varCost, whichBlock);
16    newVars.push_back(var);
17    return status;
18 }
```

Listing 2: DIP solveRelaxed for GAP example

of this using one of the many well-known algorithms for solving this problem. We could explicitly define the knapsack problem as an MILP when defining the matrix $\mathcal{Q}'$. In that case, DIP would just call the built-in MILP solver when asked to generate new extreme points of $\mathcal{P}'$. Instead, we employ a public-domain code for solving the binary knapsack problem distributed by Pisinger at [26] that uses the *combo algorithm* described in [21]. The algorithm is based on dynamic programming. In Listing 2, we show the main elements of declaring a user-specified solver for the subproblem by derivation of the base function `solveRelaxed`. The framework's recourse is determined by the `DecompSolverStatus` status code returned by this method. In the case of `DecompSolStatNoSolution`, the framework attempts to solve the subproblem as a generic MILP, assuming an explicit description of $\mathcal{Q}'$ was provided. The inputs to the method `solveRelaxed()` are as follows:

- `whichBlock` defines which block it is currently processing, and
- `costCoeff` defines the coefficients of the cost vector used in the subproblem.

In line 13, we are calling the specialized knapsack solver. This code solves the binary knapsack problem using the provided cost function. It returns a sparse vector that represents the solution to that knapsack problem. Then, in lines 15–17, that vector is used to create a `DecompVar` object that is then returned in a list of extreme points to be considered as candidates for adding to the master formulation. Listing 3 shows the main function that can be used to solve the GAP. The full source code for DIP and this example can be downloaded from [10].

GAP in DipPy. Another approach to using the DIP framework is through DipPy, an optimization modeling language that adds support for specifying decompositions to PuLP, a Python-based modeling language for linear optimization problems. The syntax of DipPy is straightforward, natural and will feel familiar to anyone who has used an algebraic modeling language. In Listing 4, we show how to construct a model for solving the GAP in DipPy. In lines 6–13, the capacity

```
1  int main(int argc, char ** argv){
2    //create the utility class for parsing parameters
3    UtilParameters utilParam(argc, argv);
4    bool doCut          = utilParam.GetSetting("doCut",      true);
5    bool doPriceCut     = utilParam.GetSetting("doPriceCut", false);
6    bool doRelaxCut     = utilParam.GetSetting("doRelaxCut", false);
7
8    //create the user application (a DecompApp)
9    GAP_DecompApp gapApp(utilParam);
10
11
12   //create the CPM/PC/RC algorithm objects (a DecompAlgo)
13   DecompAlgo   * algo = NULL;
14   if(doCut)      algo = new DecompAlgoC (&gapApp, &utilParam);
15   if(doPriceCut) algo = new DecompAlgoPC(&gapApp, &utilParam);
16   if(doRelaxCut) algo = new DecompAlgoRC(&gapApp, &utilParam);
17
18   //create the driver AlpsDecomp model
19   AlpsDecompModel gap(utilParam, algo);
20
21   //solve
22   gap.solve();
23 }
```

Listing 3: DIP main for GAP

constraints and assignment constraints are constructed. Note how the blocks are defined in a very intuitive fashion by adding the subproblems one by one to a Python list. In line 15, the subproblem solution method `relaxed_solver` is set. The definition of the `relaxed_solver` is shown in Listing 5. Due to space constraints, we do not show the definition of the knapsack oracle `knapsack01` itself, but note that it is also written in Python. In general, all customization can be done directly in Python—Python functions for solving the subproblem, generating valid inequalities, etc., will be called automatically from DIP when the problem is solved. The return value of `relaxed_solver` is a list of `DecompVar` objects having negative reduced cost. The full source code for the GAP example can be obtained in the DIP repository [10].

5 Computational Results

This section details our recent computation experience using DIP as a generic, decomposition-based solver for mixed integer linear optimization problems. In the following two sections, we give computational results that illustrate our experience with two types of models. In Section 5.1, we discuss our experience solving instances for which there is known block structure provided by the user. In Section 5.2, on the other hand, we discuss our most recent experiments involving the automatic identification of block structure for generic MILP instances that do not appear at first to have any such structure. The latter is obviously the much more difficult case and this is evident in the results. For a more detailed description of the methodology for identifying the block structure, see [36].

The results reported here are of two types. First, we illustrate the usefulness of decomposition-based methodology for solving problems with block structure when compared with a state-of-the-

```
prob = dippy.DipProblem("GAP", LpMinimize)

# objective
prob += lpSum(assignVars[m][t] * COSTS[m][t] for m, t in MACHINES_TASKS), "min"

# machine capacity (knapsacks, relaxation)
for m in MACHINES:
    prob.relaxation[m] +=
    lpSum(assignVars[m][t] * RESOURCE_USE[m][t] for t in TASKS) <= CAPACITIES[m]

# assignment
for t in TASKS:
    prob += lpSum(assignVars[m][t] for m in MACHINES) == 1

prob.relaxed_solver = relaxed_solver

dippy.Solve(prob)
```

Listing 4: DipPy createModels for GAP example

```

def relaxed_solver(prob, machine, redCosts, convexDual):
    # get tasks which have negative reduced
    task_idx = [t for t in TASKS if redCosts[assignVars[machine][t]] < 0]
    vars     = [assignVars[machine][t] for t in task_idx]
    obj      = [-redCosts[assignVars[machine][t]] for t in task_idx]
    weights  = [RESOURCE_USE[machine][t] for t in task_idx]

    z, solution = knapsack01(obj, weights, CAPACITIES[machine])
    z = -z

    # get sum of original costs of variables in solution
    orig_cost  = sum(prob.objective.get(vars[idx]) for idx in solution)
    var_values = [(vars[idx], 1) for idx in solution]

    dv = dippy.DecompVar(var_values, z-convexDual, orig_cost)

    # return, list of DecompVar objects
    return [dv]
```

Listing 5: DipPy solveRelaxed for GAP example

art commercial MILP solver using a single thread. Second, we describe our experience executing DIP on multiple cores by solving the subproblems in parallel. The parallelization was done in a straightforward and naive fashion, with the problems associated with each block assigned to processors in a round-robin fashion. For these experiments, the assignment was done statically at the outset of each iteration. This works fine for the size problems discussed here, though a better approach would be to queue the subproblems and assign them to idle processors dynamically. Each individual block was solved using CPLEX 12.2.

These experiments were performed on single nodes of a cluster, each of which has two 8 core processors, each running at 2 GHz and with 512 KB cache. These nodes have 32 GB of shared memory, though memory did not pose a limitation in any of the experiments. To assess scalability, we use the well-known statistic known as *efficiency*, defined below,

$$E = \frac{T_1}{T_p \times p}, \tag{21}$$

where T_p is defined as the wallclock time using p threads.

5.1 Structured Instances

ATM cash management problem. The first problem class we report on is an ATM cash management problem, originally described in [11]. The objective of the mixed integer model is to determine a schedule for allocation of cash inventory at bank branches to service a preassigned subset of automated teller machines (ATM). It was originally a mixed integer nonlinear programming model and was approximated by a mixed integer linear programming model. These are instances for which we know the block structure and so should be close to ideal for demonstrating the scalability of our approach to multithreaded parallelism. In the experiments below, instances available in the DIP repository [10] were solved using the ATM application also distributed with DIP.

Table 2 shows the results. The number of ATM machines in the instance determines the number of blocks and is indicated by the first number in the name (first column). To avoid redundancy, we don't show the full name, leaving off the prefix "*atm_rand*." We show results here for instances having 5, 10, and 20 blocks. Since having more threads than blocks is of no benefit, we only show results for each set of instances using a number of threads less than the number of blocks in the instance. The second column, labeled "ns", shows the normalized sample standard deviation for time required to solve each block with a single thread in the first iteration. The columns are labeled with a combination of either a "T" (time) or "E" (efficiency) and a number indicating the number of threads.

From Table 2, we can see that the efficiency decreases as the number of threads increases, as would be expected. For the largest instances, we see a drop in efficiency to approximately 0.3 for 16 threads. In general, there are two primary reasons for this drop in efficiency. One is that, since we are using the simplex algorithm to solve the master problem, this part is inherently sequential. Therefore, the time spent solving the master problem as a fraction of total running time will increase as the number of threads increases. In the case of ATM, this does not seem to be a major factor, as the fraction of time spent solving the subproblems versus the master is more than .95 for almost all instance. In cases where this is a problem, it may be possible to parallelize the LP solve using an interior point algorithm, but it is not clear this would pay dividends (see [18] for a discussion of the tradeoffs).

Table 2: Clock time for *ATM* instances from 1 to 16 threads

Instance	ns	T_1	T_2	E_2	T_4	E_4	T_8	E_8	T_{16}	E_{16}
5_50_1	0.41	354	226	0.78	197	.45	–	–	–	–
5_50_2	1.53	872	570	0.76	548	.40	–	–	–	–
5_50_3	0.55	113	88.6	0.63	71	0.39	–	–	–	–
5_50_4	0.86	8.5	6.5	0.65	4.9	0.43	–	–	–	–
5_50_5	0.88	218	157	0.69	113	0.48	–	–	–	–
10_50_1	0.60	625	375	0.83	305	0.51	208	0.37	–	–
10_50_2	1.00	65.4	38.8	0.84	26.5	0.61	23.8	0.34	–	–
10_50_3	0.68	1777	1007	0.88	771	0.57	665	0.33	–	–
10_50_4	0.96	8.3	5.9	0.70	3.7	0.56	2.7	0.38	–	–
10_50_5	1.02	8.7	5.0	0.87	4.3	0.5	2.9	0.38	–	–
20_100_1	1.35	332	221	0.75	107	0.77	79	0.52	69	0.3
20_100_4	0.96	28237	18627	0.75	9295	0.76	6055	0.58	4818	0.37
20_100_5	0.76	728	387	0.94	221	0.82	141	0.64	94	0.48

The second major reason is the high variability in the time required to solve the MILP arising from each of the blocks. When the times to solve these subproblems vary, some threads will be idle while waiting for the other threads to finish. The normalized standard deviation shown in Table 2 indicates the severity of the problem, though we emphasize that these statistics are not fully representative of the behavior across all iterations. Nonetheless, it is clear that these differences can be substantial, even in this fairly well-behaved case. This variability is the biggest contributor to losses in efficiency for the ATM case. This loss in efficiency can be reduced by parallelizing the solution of *individual blocks* using the internal parallelism of the subproblem solver (CPLEX in this case).

Wedding planner problem. The second problem we use for illustration is the *wedding planner problem* (WPP), which is described in detail in [25]. The problem is to propose a seating plan for a wedding by minimizing the sum of the maximum unhappiness of each table. The constraints include set partitioning constraints (each guest must be seated at exactly one table); table constraints (each table can only seat a limited number of attendees); and the unhappiness constraint, which linearizes the objective function, which treats the unhappiness of each table as the maximum unhappiness of all the guests at the table. The WPP is a difficult combinatorial optimization problem with a high degree of symmetry arising from the fact that the tables are indistinguishable. This symmetry contributes to large search trees when using traditional branch and cut.

We first show the competitiveness of using DIP in solving this class of problem by comparing the computation time using one thread with that of a leading commercial MILP solver, CPLEX 12.2. The results are shown in Table 3. The column labeled "CPLEX" is the running time with CPLEX on one core, while the column labeled "T_1" is the running time of DIP on one core. The column labeled "N_{DW}" is the number of nodes in the search tree for DIP and "N_{BC}" is for CPLEX. We observe that as the problem gets larger, the time consumed using CPLEX grows much faster than that by DWD in DIP. This indicates that there are important cuts that are valid for the subproblem (relaxation) that are not being generated by CPLEX. Also in terms of number of nodes explored in the search tree, is is much larger for CPLEX while quite a few instances can be solved at the root node with DIP. The WPP is a good example of a class of problems for which a straightforward generic version of DWD already shows a big improvement over a state-of-the-art branch-and-cut

Table 3: Clock time for *Wedding planner problem* instances

Guests	CPLEX	T_1	N_{BC}	N_{DW}	ns	T_2	E_2	T_4	E_4	T_8	E_8
16	0.73	2.70	1050	4	0.29	0.05	0.80	1.19	0.56	–	–
18	3.44	3.79	3448	1	0.26	0.09	0.80	1.67	0.56	–	–
19	1.32	3.12	1327	1	0.10	0.13	0.79	1.32	0.59	0.98	0.53
20	76.81	6.27	85929	7	0.24	0.23	0.86	2.75	0.57	1.93	0.54
21	59.43	5.41	57880	1	0.22	0.31	0.91	2.17	0.62	1.42	0.63
22	16.60	6.51	13376	1	0.20	0.21	0.91	2.64	0.61	1.80	0.60
23	28.01	9.78	20005	1	0.21	0.17	0.97	3.44	0.59	2.38	0.57
24	305.26	11.35	247074	1	0.2	0.11	0.87	3.60	0.78	3.58	0.52
25	112.28	9.83	82100	1	0.19	0.16	0.81	3.32	0.74	3.21	0.51
26	553.96	14.51	365498	1	0.19	0.4	0.77	5.54	0.65	4.69	0.51

code.

We also investigated DIP's parallel scalability when solving the subproblems in parallel, as we did with the ATM problems. The results are also shown in Table 3. As before, the columns are labeled with a combination of either a "T" (time) or "E" (efficiency) and a number indicating the number of threads/cores. As in the ATM problem, for a given instance, as the number of threads increases, the efficiency decreases. However, the efficiency for the WPP is higher in general than that for ATM problem. Since the time to solve the master problem is not a significant fraction of sequential running time, the improved efficiency is most likely a result of the fact that this is a problem for which the subproblems are identical. Note, however, that DIP does not yet have the ability to (directly) take advantage of the fact that the blocks (tables) are identical by collapsing the individual subproblems into a single subproblem, as is typically done. Thus, the individual blocks, while identical in structure have different objective functions rising from the different dual values on the capacity constraints for the individual tables. This is what leads to a loss in efficiency and the variability reported in Table 3. Collapsing the subproblems would in this case eliminate the opportunity for parallelization, but would allow the single collapsed subproblem to be solved in approximately the same time as solving the multiple blocks in parallel.

5.2 Automatically detected block structure

In the previous section, we provided examples to show how DIP can be used to solve instances with known structures, but we have recently begun performing experiments aimed at determining how well such structure can be identified automatically in instances that are initially unstructured. The approach used to do this in DIP is based on hypergraph partitioning and is described in [36].

The experiments discussed here illustrate the scalability that can be achieved in processing the root node in parallel. We chose to process only the root node for these experiments because solution times make it impractical to solve all problems to optimality with various numbers of threads. Furthermore, the scalability we can achieve in processing the root node is highly indicative of what we can expect to see when using the same technique to solve problems to optimality. For these experiments, we partitioned each matrix into 16 blocks and used the same decomposition for each instance across all experiments, though in practice, we could most likely adjust the number of blocks based on the available number of threads and other relevant factors.

The results are shown in Table 4. Here, we have added two additional columns. The columns labeled "LP%" and "DW%" are the gaps in the root node for the LP relaxation and DWLP,

Table 4: Multi-threaded automatic Dantzig-Wolfe Decomposition at the root node

instance	LP%	DW%	ns	T_1	P_1	E_4	P_4	E_8	P_8	E_{16}	P_{16}
protfold	35	16	6.6	133	0.28	0.36	0.12	0.23	0.42	0.1	0.74
pp08aCUTS	25	7	0.59	1.28	0.91	0.52	0.84	0.28	0.79	0.07	0.77
pp08a	62	9	0.28	0.65	0.93	0.4	0.87	0.28	0.83	0.14	0.87
pg5_34	16	0	0.63	79	0.92	0.58	0.79	0.33	0.77	0.21	0.79
machrophage	100	2	4.0	488	0.99	0.62	0.99	0.38	0.98	0.24	0.98
manna81	1	0	0.37	38	0.79	0.4	0.67	0.17	0.49	0.15	0.49
mkc	9	4	1.3	62	0.9	0.52	0.84	0.19	0.84	0.16	0.8
modglob	1	1	0.49	4.96	0.94	0.59	0.78	0.74	0.72	0.17	0.73
10teams	1	1	3.36	103	0.6	0.15	0.45	0.1	0.54	0.03	0.31
cap6000	0	0	0.68	1.43	0.67	0.52	0.44	0.29	0.36	0.19	0.28
disctom	0	0	3.3	173	0.23	0.27	0.14	0.12	0.16	0.08	0.13
fixnet6	70	20	1.31	0.73	0.88	0.35	0.78	0.26	0.72	0.15	0.68

respectively. We note that there is substantial improvement in some cases, though this result must, of course, be taken with a grain of salt, since computing times for solving these two relaxations are different. We have also added columns labeled with a "P" to indicate the fraction of time spent solving the subproblems, as it becomes more significant here.

Overall, we see the same rough trend as with the ATM and WPP instances. Computation time decreases as the number of threads increases, but we observe here that it is not always monotonic due both to much higher variability in the difficulty of solving the subproblems and a higher fraction of the time spent solving the master problem. Relative to the structured problems, the efficiency is low. Another reason for loss of efficiency is that the decomposition into 16 blocks might not actually be the most efficient decomposition for these unstructured instances. Larger numbers of blocks for problems with small and medium sizes can potentially increase the disparity of the subproblem difficulty, which eventually contributes to idle time.

6 Conclusions

We have presented here an overview of the use of the DIP framework as a generic, decomposition-based MILP solver. We also described some early computational experiments on both structured and unstructured instances. These experiments were aimed at demonstrating DIP's competitiveness with state-of-the-art solvers for certain classes of structured problems and scalability when used with multiple cores by solving the subproblems in parallel. Overall, we see promise in the approach described but it is clear that there is more work to be done before these methods become truly practical on a large scale.

References

[1] Agarwal, Y., Mathur, K., and Salkin, H. M. A set-partitioning-based exact algorithm for the vehicle routing problem. *Networks 19* (1989), 731–749.

[2] Balas, E., and Saltzman, M. Facets of the three-index assignment polytope. *Discrete Applied Mathematics 23* (1989), 201–229.

[3] Beasley, J. Lagrangean relaxation. In *Modern Heuristic Techniques for Combinatorial Optimization*, C. Reeves, Ed. John Wiley & Sons, New York, 1993.

[4] Bergner, M., Caprara, A., Furini, E., Lübbecke, M., Malaguti, E., and Emiliano, T. Partial convexification of general MIPs by Dantzig-Wolfe reformulation. In *Proceedings of the 15^{th} Conference on Integer Programming and Combinatorial* (2011), pp. 39–51.

[5] Dantzig, G. B., and Wolfe, P. Decomposition principle for linear programs. *Operations Research 8*, 1 (1960), 101–111.

[6] Fisher, M. The Lagrangian relaxation method for solving integer programming problems. *Management Science 27* (1981), 1–18.

[7] Fisher, M. Optimal solution of vehicle routing problems using minimum k-trees. *Operations Research 42*, 4 (1994), 626–642.

[8] Ford, L. R., and Fulkerson, D. R. A suggested computation for maximal multicommodity network flows. *Management Science 5* (1958), 97–101.

[9] Fukasawa, R., Longo, H., Lysgaard, J., de Aragão, M. P., Reis, M., Uchoa, E., and Werneck, R. Robust branch-and-cut-and-price for the capacitated vehicle routing problem. *Mathematical Programming 106* (2006), 491–511.

[10] Galati, M., and Ralphs, T. DIP. Available from `http://www.coin-or.org/projects/Dip.xml`.

[11] Galati, M. V. *Decomposition in Integer Programming*. PhD thesis, Lehigh University, 2009.

[12] Gamrath, G. Generic branch-cut-and-price. Master's thesis, Technische Universität Berlin, 2010.

[13] Geoffrion, A. Lagrangian relaxation for integer programming. *Mathematical Programming Study 2* (1974), 82–114.

[14] Gilmore, P. C., and Gomory, R. E. A linear programming approach to the cutting stock problem. *Operations Research 9* (1961), 849–859.

[15] Guignard, M., and Rosenwein, M. An improved dual-based algorithm for the generalized assignment problem. *Operations Research 37* (1989), 658–663.

[16] Held, M., and Karp, R. M. The traveling salesman problem and minimum spanning trees. *Operations Research 18* (1970), 1138–1162.

[17] Jünger, M., and Thienel, S. Introduction to ABACUS—a branch-and-cut system. *Operations Research Letters 22* (1998), 83–95.

[18] Koch, T., Ralphs, T., and Shinano, Y. Could we use a million cores to solve an integer program? *Mathematical Methods of Operations Research 76* (2012), 67–93.

[19] Lemaréchal, C. Lagrangean relaxation. In *Computational Combinatorial Optimization*, M. Jünger and D. Naddef, Eds. Springer, New York, 2001, pp. 112–156.

[20] Lucena, A. Steiner problem in graphs: Lagrangean relaxation and cutting planes. *COAL Bulletin 28* (1992), 2–8.

[21] Martello, S., Pisinger, D., and Toth, P. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Manage. Sci. 45*, 3 (1999), 414–424.

[22] Mehrotra, A., and Trick, M. A column generation approach to graph coloring. *INFORMS Journal on Computing 8*, 4 (1996), 344–354.

[23] Neame, P. J. *Nonsmooth Dual Methods in Integer Programming*. PhD thesis, University of Melbourne, 1999.

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

[25] O'Sullivan, M., Lim, Q., Walker, C., Dunning, I., and Mitchell, S. Dippy: A simplified interface for advanced mixed integer programming. Report 685, University of Auckland Faculty of Engineering, 2011.

[26] Pisinger, D. David Pisinger's optimization codes. Available from `http://www.diku.dk/hjemmesider/ansatte/pisinger/codes.html`.

[27] Ralphs, T., Kopman, L., Pulleyblank, W., and Trotter Jr., L. On the capacitated vehicle routing problem. *Mathematical Programming 94* (2003), 343–359.

[28] Ralphs, T., and Ladányi, L. *COIN/BCP User's Manual*, 2001. Available from `http://www.coin-or.org`.

[29] Ralphs, T. K., and Galati, M. V. Decomposition in integer programming. In *Integer Programming: Theory and Practice*, J. Karlof, Ed. CRC Press, 2005, pp. 57–110.

[30] Ralphs, T. K., and Galati, M. V. Decomposition and dynamic cut generation in integer programming. *Mathematical Programming 106* (2006), 261–285.

[31] Savelsbergh, M. A branch-and-price algorithm for the generalized assignment problem. *Operations Research 45*, 6 (1997), 831–841.

[32] van den Akker, J., Hurkens, C., and Savelsbergh, M. Time-indexed formulations for machine scheduling problems: Column generation. *INFORMS Journal on Computing 12* (2000), 111–124.

[33] Vance, P. H., Barnhart, C., Johnson, E. L., and Nemhauser, G. L. Solving binary cutting stock problems by column generation and branch and bound. *Computational Optimization and Applications 3* (1994), 111–130.

[34] Vanderbeck, F. Computational study of a column generation algorithm for bin packing and cutting stock problems. *Mathematical Programming 86*, 3 (1999), 565–594.

[35] Vanderbeck, F. BapCod: A generic branch-and-price code, 2012. Available from `http://ralyx.inria.fr/2007/Raweb/realopt/uid31.html`.

[36] Wang, J., and Ralphs, T. K. Computational experience with hypergraph-based methods for automatic decomposition in discrete optimization. In *Proceedings of the Conference on Constraint Programming, Artificial Intelligence, and Operations Research* (2013), pp. 394–402.

[37] Xu, Y., Ralphs, T. K., Ladányi, L., and Saltzman, M. J. Alps: A framework for implementing parallel search algorithms. In *The Proceedings of the Ninth INFORMS Computing Society Conference* (2005), pp. 319–334.