Ann Oper Res (2006) 142: 99–118
DOI 10.1007/s10479-006-6163-1

A management system for decompositions in stochastic programming

Robert Fourer · Leo Lopes

Abstract This paper presents two contributions: A set of routines that manipulate instances of stochastic programming problems in order to make them more amenable for different solution approaches; and a development environment where these routines can be accessed and in which the modeler can examine aspects of the problem structure. The goal of the research is to reduce the amount of work, time, and cost involved in experimenting with different solution methods.

1. Introduction

Stochastic programming problems are most commonly solved by creating deterministic equivalent linear programs, whose special structure has been extensively exploited by many different solution techniques. In some special instances, the modeler can determine *a priori* which technique is most likely to be successful. But as of yet, no technique or implementation has been shown to consistently outperform the others in the general case. In addition, issues related to computer performance, solver availability, and numerical stability further complicate the problem of selecting a solution technique.

Ultimately, in many cases, the current best approach is simply to experiment with different techniques until one is found that works well. Unfortunately, different techniques require different descriptions of the same problem instance to perform at their best. Producing alternate descriptions using previously available tools requires significant effort to re-code the model and the data acquisition routines. This cost detracts from the attractiveness of a stochastic optimization approach.

R. Fourer
Industrial Engineering and Management Sciences, Northwestern University, 2145 N. Sheridan Rd. #C234, Evanston, IL 60208
e-mail: 4er@iems.nwu.edu

L. Lopes (✉)
Systems and Industrial Engineering, University of Arizona, 1127 East North Campus Drive, Tucson, AZ, 85719
e-mail: leo@sie.arizona.edu

In this research we examine the problems brought about by differences in structure required by different solution techniques in stochastic programming. This research and the software described in it address these problems in two steps: first, by creating a set of routines that convert instances of stochastic optimization problems to the representations useful to the most common solution techniques; second, by integrating those routines into an environment which allows the modeler to access them easily and quickly, and which provides interactive information about the problem structure. The motivation of the work is to reduce the cost of the stochastic programming approach, by allowing for the first time for experimentation with different solution methods without any recoding effort. It is this reduction in cost that is the fundamental contribution of the research.

In particular, the following techniques are covered: the generation of Deterministic Equivalents, both explicit and implicit; Benders decomposition, including stage aggregation; and lagrangian relaxation. In addition, the software environment provides a sparsity structure browser and a scenario tree browser, to aid the modeler in understanding the algebraic and stochastic structure present in the problem.

Our system uses a database for back-end storage. Instances can be created directly in the database, or provided using SMPS (Birge et al., 1987; Gassmann and Schweitzer, 1996) files. After manipulation, the results can also be written as SMPS files or obtained from the database using appropriate routines supplied as part of the research.

The research described here is similar to some aspects of SLP-IOR (Kall and Mayer, 1996). Both systems consider the issue of different formulations for different solvers, have a specific focus on usability, and provide features for structure analysis. The principal distinction between the two is that (Kall and Mayer, 1996) covers the entire life cycle of the stochastic programming model, while our system focuses on the phase after model definition and before solution. As a consequence, our system has richer manipulation features. The SPInE (Valente et al., 2001) system, also a full life-cycle system, describes similar structure manipulation routines to ours, but lacks a stage aggregator. It also includes a similar, albeit less flexible, graphical feedback mechanism. Our stage aggregation procedures are inspired by those in Dempster and Thompson (1998), but we make them available to any solver that can read SMPS or that uses our library, and make less restrictive assumptions about problem structure than those in that research. Other systems incorporating some of the functionality in this system (and other unrelated functionality) include Stochastics (Dempster, Scott, and Thompson, 2002), SP/OSL (King, 1994), and SETSTOCH (Condevaux-Lanloy and Fragniere, 1998). Finally, our system differs in design from the above systems in modularity. Every major component of our system, including each structure manipulator, the GUI and the parsers can be used independently from the others and as parts of other systems.

The paper is organized as follows: In section 2 we lay out our notation and briefly explain at a very high abstraction level the decomposition methods we addressed, in order to explain *what* it is exactly that our system provides in each case. In Sections 3, 4 and 5 we explain in detail *how* the structure manipulation is accomplished for each solution technique. Section 6 describes the visual environment in which the routines are accessed. Finally, in Section 7 we talk about some conclusions, opportunities for improvement and further research.

2. Decomposition methods in stochastic programming

Decomposition approaches have a very long history, almost as long as linear programming itself. They are particularly prevalent in stochastic programming, given the size of the problem instances involved. Many instances of stochastic programming problems are not only too large to *solve* without decomposition, but too large even to *fit in memory* without decomposition, and likely to remain so for a long time, even at the current pace of computing power development. For this research, we consider two of the most common forms of decomposition approaches: the L-shaped method (Van Slyke and Wets, 1967), based on Benders decomposition; and lagrangian relaxation.

In this section, we provide some details about the problem itself, and about the two decomposition methods above.

2.1. Problem description

Stochastic Programming problems in general are simply optimization problems in which some subset of the data is not assumed to be known with certainty. Instead, this subset of the data is assumed to be sufficiently well described by some random variable. For example: most inventory models implemented and in use in the business environment today are formulated as if demands were constant. These models usually employ expected values for demands. But in fact, in most situations, demands are not constant, although they can often be well described by some probability distribution. When there is value to be added in hedging against different realizations of the random variables, a stochastic programming approach is likely to be profitable.

In this research, we consider the case in which the random variables have discrete distributions or have been approximated by discrete distributions, so that each combination of outcomes of all the random variables defines a scenario s, with associated probability p_s. Let the set of all scenarios be $\mathcal{S}$. Under each scenario $s \in \mathcal{S}$, all the parameters are known.

The objective is to minimize some current, known cost of the decisions x_1 taken now, plus the *recourse function*, a measure of future cost, which may depend on: the current decision x_1 (which can not be changed later); future decisions x_2 (made after observing the uncertainty), and the outcomes of the random variables under each scenario s.

We can now write the problem as:

$$\begin{array}{llll}
\min & c^T x_1 + \sum_{s \in \mathcal{S}} p_s q_s^T x_{2,s} & & \\
& A x_1 & & = b \\
& T_s x_1 & + W_s x_{2,s} & = h_s, \forall s \in \mathcal{S} \\
& & A_s x_{2,s} & = b_s, \forall s \in \mathcal{S} \\
& x_1 \geq 0 & & \\
& & x_{2,s} \geq 0 \ \forall s \in \mathcal{S} &
\end{array}$$

The T_s matrix is referred to as the *technology* matrix. The W_s matrix is referred to as the *recourse* matrix.

At discrete, known, points in time, a subset of the random variables is observed. For example, a stage might be a fiscal quarter, and a set of random variables of interest might be the demand for a product during each fiscal quarter. The interval between points in which new information becomes available is referred to as a *stage*. For simplicity, the exposition above used two-stage problems. Multi-stage problems, which are covered by this research, can be defined recursively in terms of two-stage problems.

If all the scenarios in a subset $\mathcal{A} \subseteq \mathcal{S}$ share the same outcomes for all random variables observed up to some stage t, then all the data under those scenarios, and the decisions made under those scenarios, must be the same up to that point. We call such subsets *bundles*, following (Rockafellar and Wets, 1991). This constraint on the decision process, referred to as *non-anticipativity*, is a key concept in stochastic programming. Non-anticipativity prevents a decision being taken now from using information that will only become available in the future. This is what makes the stochastic programming approach so realistic, and is what makes the resulting computational problems so difficult to solve.

Bundles are all disjoint at any stage t, and singletons at the last stage. The set of all bundles at each stage t is denoted Λ_t (Λ_t is a *partition of* $\mathcal{S}$ at stage t).

The *scenario tree* is a data structure representing the relationships between different bundles (and consequently scenarios). A scenario is represented in the tree by a path from the root to a unique leaf of the tree. Thus we label each leaf with the name of its associated scenario. Example scenario trees can be found in Figure 1. In the leftmost tree in that figure, $\Lambda_2 = \{\{A, B, C\}, \{D, E\}\}$.

By using scenarios, we can create very large scale linear programs whose solution is the same as that of the stochastic program. These Deterministic Equivalents (DE) have a very peculiar algebraic structure, which can be exploited by different decomposition methods.

There are two ways of writing DEs: the implicit and the explicit (also known as split-variable) formulations. They differ in the way non-anticipativity is handled. Before we can provide the precise algebraic formulations, we need one more definition. We need a relation that maps a bundle in stage t to the bundles in stage $t + 1$ which are composed of the same scenarios (the children of a node in the scenario tree). For each bundle $\mathcal{A} \in \Lambda_t$, let:

$$U(\mathcal{A}) = \{\mathcal{B} \in \Lambda_{t+1} | \mathcal{B} \subseteq \mathcal{A}\}$$

For example, $U(\{A...C\}) = \{\{A\}, \{B\}, \{C\}\}$ in the leftmost tree in Figure 1.

In implicit DEs, there is a single set of variables $x_{t,\mathcal{A}}$ associated with each bundle $\mathcal{A}$ at any stage t. Constraints then link these variables to the various bundles that branch from $\mathcal{A}$ in stage $t + 1$. This generates the following general algebraic representation, in which $p_{\mathcal{A}}$ is the *un*conditional probability of the realization of a bundle (the probability that any of the scenarios in the bundle is realized):

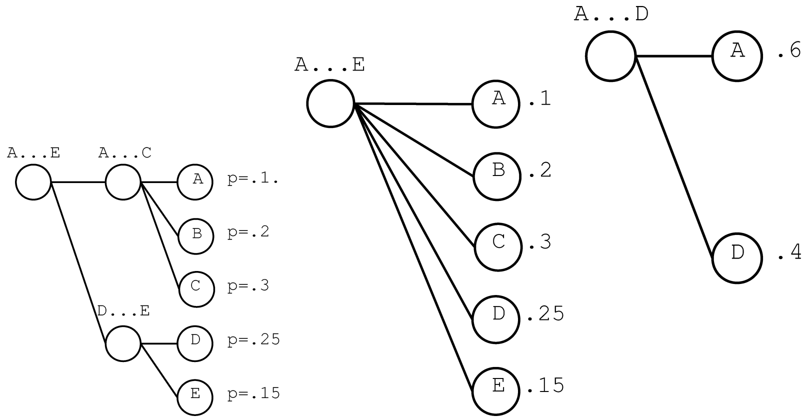


Fig. 1 A scenario tree for a 3-stage problem, and the result of the possible aggregations

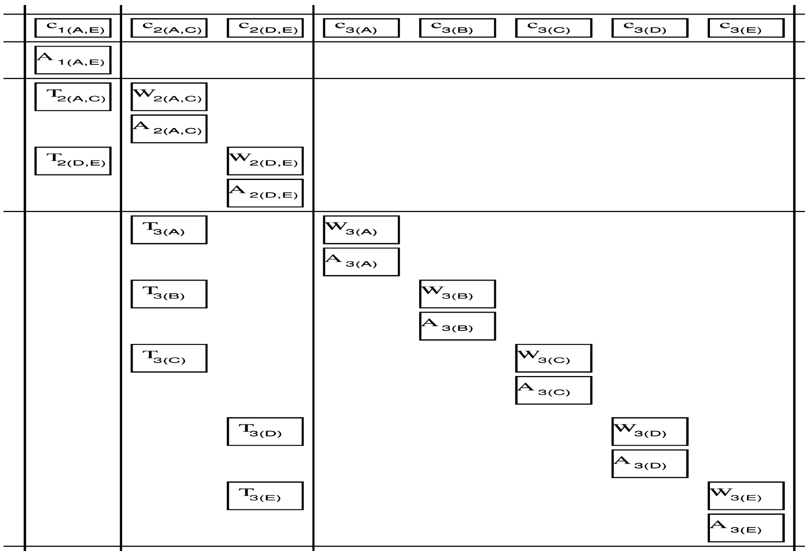


Fig. 2 The algebraic structure of a 3-stage problem with a tree like the one in Figure 1 using the *implicit* formulation

$$
\begin{array}{llll}
\min & c_1x_1 + \sum_{t\in\{2,\dots,T\}} \sum_{\mathcal{A}\in\Lambda_t} p_{\mathcal{A}} q_{t,\mathcal{A}} x_{t,\mathcal{A}} & & \\
& A_1x_1 = b_1 & & \\
& A_{t,\mathcal{A}}x_{t,\mathcal{A}} = b_{t,\mathcal{A}} & \forall t\in\{2,\dots,T\}, \forall\mathcal{A}\in\Lambda_t & \\
& T_{t,\mathcal{B}}x_{t-1,\mathcal{A}} + W_{t,\mathcal{B}}x_{t,\mathcal{B}} = h_{t,\mathcal{B}} & \forall t\in\{2,\dots,T\}, \forall\mathcal{A}\in\Lambda_{t-1}, \forall\mathcal{B}\in U(\mathcal{A}) & \\
& x_1 \geq 0 & & \\
& x_{t,\mathcal{A}} \geq 0 & \forall t\in\{2,\dots,T\}, \forall\mathcal{A}\in\Lambda_t &
\end{array}
\tag{2.1}
$$

Figure 2 illustrates the sparsity structure for the leftmost tree in figure 1 using an implicit DE.

The corresponding explicit (or split-variable) DE would have the following formulation:

$$
\begin{array}{lll}
\min & \sum_{t\in\{1,\dots,T\}} \sum_{s\in\mathcal{S}} p_s q_{t,s} x_{t,s} & \\
& A_{t,s}x_{t,s} = b_{t,s} & \forall t\in\{1,\dots,T\}, \forall s\in\mathcal{S} \\
& T_{t,s}x_{t-1,s} + W_{t,s}x_{t,s} = h_{t,s} & \forall t\in\{2,\dots,T\}, \forall s\in\mathcal{S} \\
& x_{t,s_1} - x_{t,s_2} = 0 & \forall t\in\{1,\dots,T\}, \forall\mathcal{A}\in\Lambda_t, \\
& & \text{choose } s_1\in\mathcal{A}, \forall s_2\in\mathcal{A}\setminus\{s_1\} \\
& x_{t,s} \geq 0 & \forall t\in\{1,\dots,T\}, \forall s\in\mathcal{S}
\end{array}
\tag{2.2}
$$

Figure 3 represents the sparsity structure of the corresponding explicit DE for the same tree as Figure 2. In this formulation, the decision variables in each scenario are duplicated throughout all stages, even though, up to stage t, they must take the same values for each bundle of scenarios that can not be distinguished up to stage t. To force the variables to take the same values, explicit non-anticipativity constraints are added. They are represented in the figure by the crossed boxes. This formulation, then, has more variables and more constraints

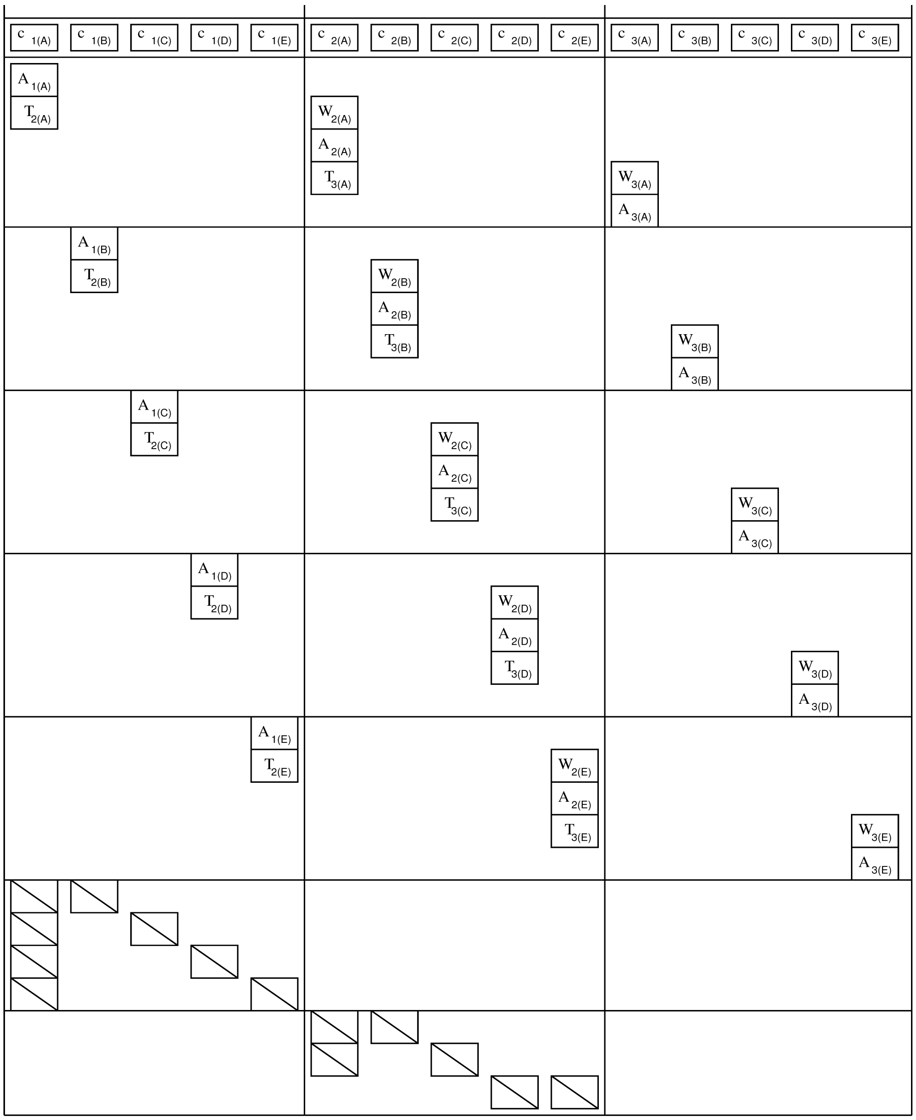


Fig. 3 The *explicit* deterministic equivalent for the tree in Figure 1 with non-anticipativity constraints represented by diagonal boxes

than the implicit DE, but is useful when the underlying problem has some special structure which would be destroyed if the implicit DE were used.

It is also useful to think of a subproblem deterministic equivalent (SDE), as the DE generated starting at some stage $t > 1$ for a particular bundle. These SDEs come up when discussing some types of decomposition schemes.

Our system provides the ability to generate both explicit and implicit DEs. This allows any stochastic problem (limited of course to the computer resources available to the modeler) to be solved by general-purpose linear programming solvers.

2.2. Benders decomposition

The most common decomposition approach used in stochastic programming is Benders decomposition and its improvements, referred to in the stochastic programming context as the L-shaped method (Slyke and Wets, 1969). This outer linearization scheme works on the implicit formulation in the following way (For simplicity we will use a 2-stage problem here. But the method is extendible to multistage by being applied recursively):

1. The problem is rewritten into two:
   - A first stage problem with an approximation of the recourse function (initially $f(x_1) = 0$):

$$\begin{aligned} \min_{x_1}\ & c_1 x_1 + f(x_1) \\ & A_1 x_1 \quad = b_1 \\ & x_1 \geq 0 \end{aligned} \qquad (2.3)$$

   - A second stage problem as a function of specific values of the first stage variables. This problem is now decomposable into $|\Lambda_2|$ subproblems. For each $\mathcal{A} \in \Lambda_2$, the subproblem would be:

$$\begin{aligned} \min_{x_{2,\mathcal{A}}}\ & p_{\mathcal{A}} q_{2,\mathcal{A}} x_{2,\mathcal{A}} \\ & A_{2,\mathcal{A}} x_{2,\mathcal{A}} = b_{2,\mathcal{A}} \\ & W_{2,\mathcal{A}} x_{2,\mathcal{A}} = h_{2,\mathcal{A}} - T_{2,\mathcal{A}} x_1 \\ & x_{2,\mathcal{A}} \geq 0 \end{aligned} \qquad (2.4)$$

2. The following procedure is iteratively repeated until an adequate solution is found:
   (a) The first stage problem 2.3 is solved using the current approximation to the recourse function (initially $f(x) = 0$).
   (b) The first stage solution obtained from step (a) is tested against problems derived from each second stage subproblem 2.4.
   (c) If the current approximation is correct at that solution, we stop. Otherwise, dual information from the subproblems is used to generate a cut that effectively improves the approximation $f(x)$, and another loop is executed.

For a detailed exposition of the L-shaped method, see (Birge and Louveaux, 1997). The original development of Benders decomposition can be found in Benders (1962). Papers specializing it for stochastic programming can be found in Van Slyke and Wets (1967), Birge (1985). The most widely available implementations are MSLiP (Gassmann, 1990), which can be used directly from the NEOS server (Czyzyk, Mesnier, and Moré, 1998) at `http://www-neos.mcs.anl.gov/neos`, and and SP/OSL (King), which can be obtained at `http://www-3.ibm.com/software/data/bi/osl`.

The L-shaped method has two characteristics that make it an excellent candidate for parallelization using distributed computing architectures: The independence of the second stage subproblems obtained as a result of the decomposition; and the fact that the information being passed between the first stage and second stage is relatively simple. Depending on the characteristics of the computer architecture, it may be profitable to split the stage structure at different points, to maximize the use of each individual computing node. For example, if the original problem had 10 stages, we may wish to use Benders decomposition at the third, fifth, and seventh stage, and deal with the other stages by solving SDEs. Dempster and Thompson (1998) illustrates the benefits of this approach. In that system, the solver has a parameter which permits the modeler to specify the stages in which to decompose.

Most stochastic programming decomposition schemes fit well into a Master-Worker environment using a library like the one in Goux et al. (2000). Linderoth and Wright have developed systems for stochastic optimization based on the Master-Worker concept (Linderoth and Wright, 2001) for Benders decomposition, taking advantage of idle time in machines spread over wide area networks.

This manipulation can also be desirable even without parallel capability, to achieve several objectives including:

- Reducing the number of cuts being generated.
- Providing alternative iteration paths in cases where the original path leads to numerical instability.
- Maximizing the use of available memory, when the subproblems are very large.
- Minimizing disk access, when the subproblems are very small.

Unfortunately, this functionality is not commonly available in solvers, and we don't expect that to change, since adding it is not trivial, and it lies in between the modeling and solving domains. Our system provides the functionality so that the decision of where to split the decomposition becomes available to any solver that implements the L-shaped method. In fact, it is transparent to the solver, since it is done before the solver even sees the instance.

2.3. Lagrangian relaxation

Lagrangian relaxation is a technique used when difficult problems can be turned into easy problems by eliminating a subset of the constraints. Penalties for violating the constraints are introduced into the objective. These penalties are then adjusted in an attempt to *price* the constraint violation, and the problem is solved again. The objective value of the solution to the pricing problem provides a bound to the original problem. For an excellent high-level introduction to lagrangian relaxation, see (Fisher, 1985); and for a more detailed exposition, see (Fisher, 1981), by the same author.

In our case, we relax the non-anticipativity constraints. If they are written using formulation 2.2, as illustrated in Figure 3, then we will get a "reference" scenario s_1 within each bundle, and a straightforward interpretation for the penalties is available: they represent the price to be paid in each bundle for deviating from the reference scenario. If we denote the penalties by λ, we obtain:

$$\begin{array}{llll}
\min & \displaystyle\sum_{t\in\{1,\dots,T\},s\in\mathcal{S}} p_s q_{t,s} x_{t,s} + & & \\
 & \displaystyle\sum_{t\in\{1,\dots,T\},\forall\mathcal{A}\in\Lambda_t,s_1\in\mathcal{A},\forall s_2\in\mathcal{A}\setminus\{s_1\}} \lambda_{s_1,s_2}(x_{t,s_1} - x_{t,s_2}) & & \\
 & A_{t,s}x_{t,s} = b_{t,s} & \forall t\in\{1,\dots,T\}, \forall s\in\mathcal{S} & \\
 & T_{t,s}x_{t-1,s} + W_{t,s}x_{t,s} = h_{t,s} & \forall t\in\{2,\dots,T\}, \forall s\in\mathcal{S} & \\
 & x_{t,s} \geq 0 & \forall t\in\{1,\dots,T\}, \forall s\in\mathcal{S} &
\end{array} \qquad (2.5)$$

Note that with a small amount of algebraic reshuffling of the objective, this problem can be written independently for each scenario. For each scenario $s \in \mathcal{A}$, the formulation would be:

$$
\begin{aligned}
\min \sum_{t\in\{2,\ldots,T\}} &(p_s q_{t,s} + f_{t,s})x_{t,s} \\
A_{t,s}x_{t,s} &= b_{t,s} \quad \forall t \in \{1,\ldots,T\} \\
T_{t,s}x_{t-1,s} + W_{t,s}x_{t,s} &= h_{t,s} \quad \forall t \in \{2,\ldots,T\} \\
x_{t,s} &\geq 0 \quad \forall t \in \{1,\ldots,T\}
\end{aligned}
\tag{2.6}
$$

where :

$$
f_{t,s} = \begin{cases} \sum_{s_2 \in \mathcal{A}\setminus\{s\}} \lambda_{s,s_2} & \text{if } s \text{ is the reference scenario in } \mathcal{A} \in \Lambda_t \\ -\lambda_{s_1,s} & \text{if } s_1 \text{ is the reference scenario} \end{cases}
$$

Given a set of prices for violating the constraints, the solution to 2.5 provides a lower bound to the objective of the original problem. Thus, we obtain the best bound by *maximizing* 2.5 over all λ. Lagrangian relaxation is a very popular technique in integer programming, because the bounds it provides are often much better than those of the linear programming relaxation, and can be shown to be at least as good in the worst case. It is an especially useful technique when at least one of the following two conditions are met:

- The subproblems 2.6 can be solved by some simple problem-specific procedure.
- The prices have some meaning to the application, and can be found easily, or can be obtained or guessed from characteristics of the application.

Unfortunately, in the general case, where neither of the above considerations are present, convergence would likely be slow. Furthermore, obtaining the solution to the original problem from the solution to the pricing problem would not be trivial. Thus, lagrangian relaxation-based methods are usually employed when problem-specific information about the problem is available. This knowledge may be at the modeling level, at the algebraic structure level, or both. In the context of stochastic programming, this has mostly been done in the energy sector. Another application for lagrangian relaxationin stochastic programming, which is not problem-specific, is in stochastic integer programming (Carøe and Schulz, 1999).

Our approach to supporting lagrangian relaxation is to generate automatically those components of the lagrangian procedure which can be deduced from the problem structure. We generate the relaxed subproblem (the pricing problem); and generate a subroutine which given a set of prices returns the objective coefficients for the pricing problem.

3. Direct solution methods

When it is possible to store the entire problem in memory, it is often advantageous to solve the DE directly. Sometimes, solving the DE directly is the only available option. Furthermore, generating DEs in several different ways is essential to many aspects of this research. In this section, we will introduce some aspects of the SMPS format, the current standard for instances of stochastic programs, that are relevant to our research. We then explain aspects of how we generate deterministic equivalents.

3.1. The SMPS format

The SMPS format (Birge et al., 1987) is an extension of the MPS format originally proposed by IBM for communicating instances of linear programming problems. SMPS is the main method for communicating and storing instances of stochastic programming problems today. In SMPS a stochastic programming problem is assumed to be an extension of a deterministic problem where expected values have been replaced by distribution information. This assumption was important to provide backward compatibility with MPS, among other reasons. The constructs in SMPS supporting scenarios assume that all scenarios at each stage have the same sparsity structure. Newer proposed amendments to SMPS (Gassmann and Schweitzer, 1996) contain constructs that would drop this assumption, but they are not yet implemented in the available solvers.

With what is available today, an arbitrary scenario is chosen as a "root" scenario, and a list of all the nonzeroes for this scenario is provided. Scenarios then branch from each other, such that only the coefficients which differ between scenarios need to be stored. Each scenario is provided by specifying four data items:

- Its parent.
- The stage in which it branches from its parent.
- Its path probability.
- A list of elements in all the relevant matrices that make this scenario different from its parent.

For more information on SMPS, a very good reference is the web documentation at `http://www.mgmt.dal.ca/sba/profs/hgassmann/SMPS2.htm`.

3.2. Generating DEs from SMPS

The algorithms for generating DEs from a stochastic problem and a scenario tree such as one may find in an SMPS file are simple and well understood. They have been implemented numerous times in solvers and other tools. Thus we describe here only the aspects of our implementation that are unique.

Our notation varies slightly from that of other stochastic programming papers. In many papers, the A_t matrix for $t > 1$ is not explicitly represented but is instead considered part of the W_t matrix. That notation is sufficient to describe decomposition algorithms. In the context of structure manipulation, on the other hand, it is convenient to distinguish between the two matrices. W_{t+1} represents (together with T_{t+1}, which actually refers to decisions in stage t), the constraints on the decisions at stage $t+1$ imposed by the decisions at stage t. A_{t+1} represents constraints on decisions at stage $t+1$ which are independent of any past or future decisions. Thus, all the operations involving generation of new scenario trees, or coupling of aggregated stages, among others, involve the row count of W_{t+1}, and not the row count of the block formed by $\left[\begin{smallmatrix} W_{t+1} \\ A_{t+1} \end{smallmatrix}\right]$. Although in SMPS this block is not allowed to have a different number of rows for different scenarios at the same stage, there is no mathematical reason for such a restriction. Furthermore, it may be very useful from the modeling perspective to allow the number of constraints to vary from scenario to scenario. For example, it could be that under a certain scenario, a government regulation comes into effect and must be observed in the decision process. In our system, we have allowed the number of rows in different scenarios at the same stage to vary.

In SMPS, the number of rows and columns of all matrices is the same for all the scenarios in the same stage. Thus, to generate the DE, the appropriate blocks (including constraint senses) from the "core" section are copied and used as a frame (providing the sparsity structure and default values) for each node v in stage t, and each of its children $u_1, \ldots, u_k$. One set of columns is copied from stage t, and k sets of columns and rows are copied from stage $t + 1$.

4. Benders decomposition and stage aggregation

The functionality we provide in the context of Benders decomposition is stage aggregation. This consists of rewriting the algebraic structure of the problem so that the new problem has fewer stages than the original one.

The operation consists, at the most abstract level, of transforming a problem whose scenario tree looks like the leftmost one in Figure 1 into one whose scenario tree looks like the middle tree, if stages 1 and 2 are aggregated; or like the rightmost one, if stages 2 and 3 are aggregated. If all stages are aggregated, we have the deterministic equivalent.

As a consequence of using the SMPS data structures, especially of the assumption of only one frame for all scenarios at a given stage, many of the operations needed for stage aggregation become quite a bit more complicated than they would be if done at the solver level. The main advantage of doing them at this level, before the problem reaches the solver, is that now every solver can take advantage of the feature, without even being aware of it.

What we are accomplishing, essentially, is writing part of the problem as an SDE, and adjusting the rest of the problem accordingly. This is the approach in Dempster and Thompson (1998), where computational results illustrate the usefulness of the approach. Notice that this stage aggregation is different from the approach in Klaassen (1998), which actually reduces the size of the problem by aggregating data based on peculiarities of the stochasticity in the problem.

4.1. Transforming the algebraic structure

In SMPS, a single *core* problem is specified from stage 1 to stage T. Subsequently, other scenarios are specified as branching from that core. Because of this, if two nodes at stage t have different degrees (number of children), then an SDE starting at stage t must be produced from the node with the largest degree. Say for example that stages 2 and 3 are aggregated on the left tree in Figure 1, resulting in the right tree. Figure 4 illustrates the necessary manipulations. Notice that bundle $\{A, B, C\}$ has 3 children at stage 2, while bundle $\{D, E\}$ has 2. Thus the SDEs generated would be different. But to communicate this using SMPS, the SDE obtained from $\{A, B, C\}$ has to be used as the frame, and the extra node has to be zeroed out using "phantom" rows and columns when scenario D is defined.

If the last aggregated stage $t_l \neq T$, then the T_{t_l+1} matrices require special care. The problem can be better understood from Figure 5. In the DE on the left, obtained from a 3-stage problem, decomposing at the end of the first stage creates two independent two-stage problems (highlighted in different shades). Assuming that stage 2 has n variables, each of the T_3 matrices will also have n columns. In the DE on the right, in which stages 1 and 2 have been aggregated, new variables were created in the core for each bundle at stage 2. Now the T_2 matrices (which replace the original T_3 matrices) have $2n$ variables. But only half of those columns should actually be considered for any stage 2 (old stage 3) bundles. The others need to be zeroed out.

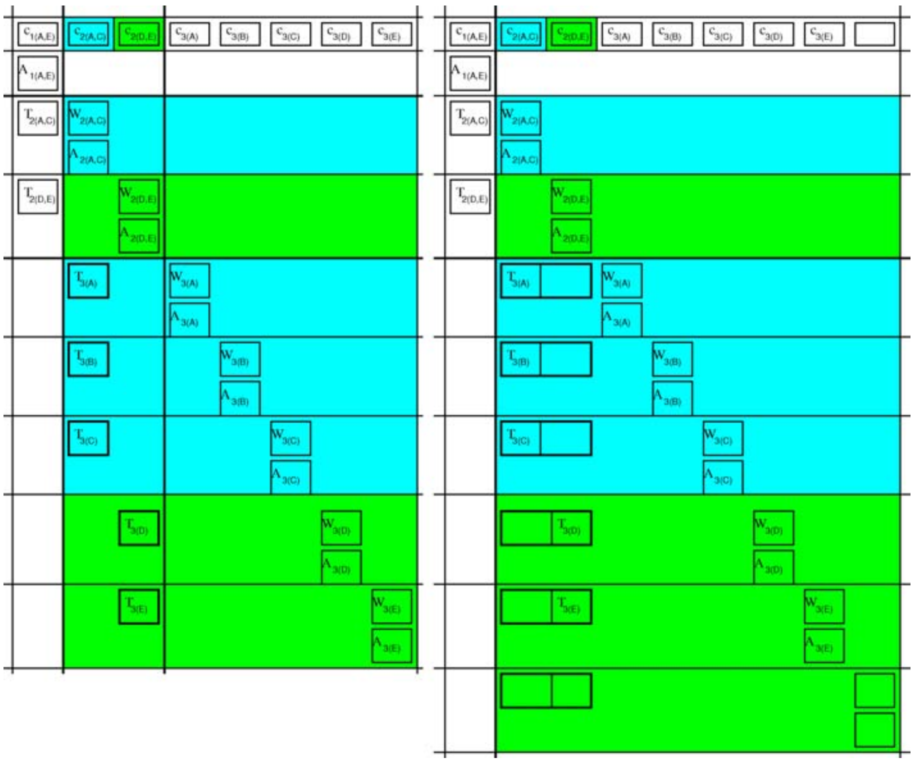


Fig. 4 Aggregation of non-balanced trees illustrating the need for phantom rows and columns

In summary, the following manipulations of the problem structure are necessary under different conditions to do the transformation above the solver level, so that the solver can create the DE correctly.

- The assignment of variables to stages needs to be redone.
- New rows and variables need to be created for the part of the problem which is being rewritten as an SDE.
- The coefficients of the objective function need to be recalculated to reflect joint probabilities.
- A new framing scenario tree needs to be calculated in a way that minimizes empty scenarios, and empty scenarios need to be zeroed out.
- For an aggregation ending at stage t_l, the T_{t_l+1} matrices need to be adapted so that they only refer to their parent.

4.2. Amending the scenario tree

In order to find the sparsity structure (or *core*, in SMPS parlance) that minimizes the number of scenarios that need to be zeroed out, a Smallest Common Embedded Supertree (SCES) problem must be solved. Here is a statement of the problem, generalized from Gupta and Nishimura (1998):

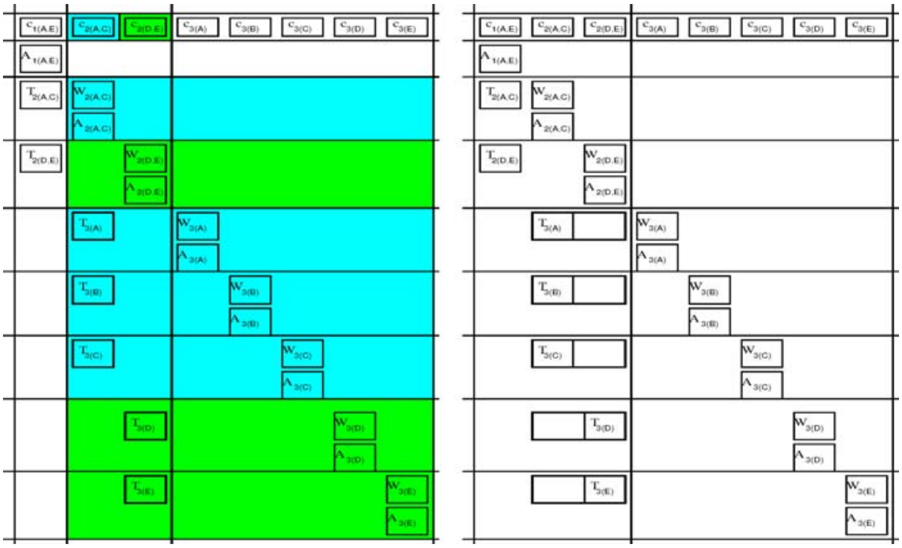


Fig. 5 T_{t_l+1} matrices become wider after the aggregation, and need to be filled with zeroes where appropriate

Smallest common embeddable supertree problem (SCES): Given a forest F, determine a new tree (V^*, E^*) minimizing $|V^*|$ such that $\forall(V, E) \in F$, (V, E) is embeddable into (V^*, E^*).

The SCES can be solved in polynomial time (Gupta and Nishimura, 1998) for two trees, but its complexity for the case with more than two trees is open. The polynomial algorithms for the SCES with two trees which we have studied rely on solving assignment problems. Generalizations of the assignment problem can not, in general, be solved in strictly polynomial time. Also, the associative property does not apply to the two-tree SCES operation. This leads us to believe that the problem is hard for more than two trees.

In the stochastic programming case, the trees have two convenient properties: All the trees have the same depth (the longest distance from a leaf to the root is the same); and each tree is complete (every leaf is at the same depth). As part of this research, we implemented a Prolog routine that solves the problem by a smart complete enumeration, but that routine turned out, as expected, to be practical only for trees with very few nodes, since the number of complete trees with n nodes is exponential in n.

All the trees we could find in test problems were depth-full: all nodes at the same depth had the same degree. So we created a heuristic for the SCES that is optimal for depth-full trees, and performs well for the general case. The heuristic works as follows:

Given: A forest F of complete trees with the same maximum depth.

1. Create a supertree $(\bar{V}, \bar{E})$ with the following characteristics:

- Every node at depth d has degree

$$\max_{(V_i,E_i)\in F} \; \max_{\{v\in V_i|\text{depth}(v)=d\}} \text{degree}(v)$$

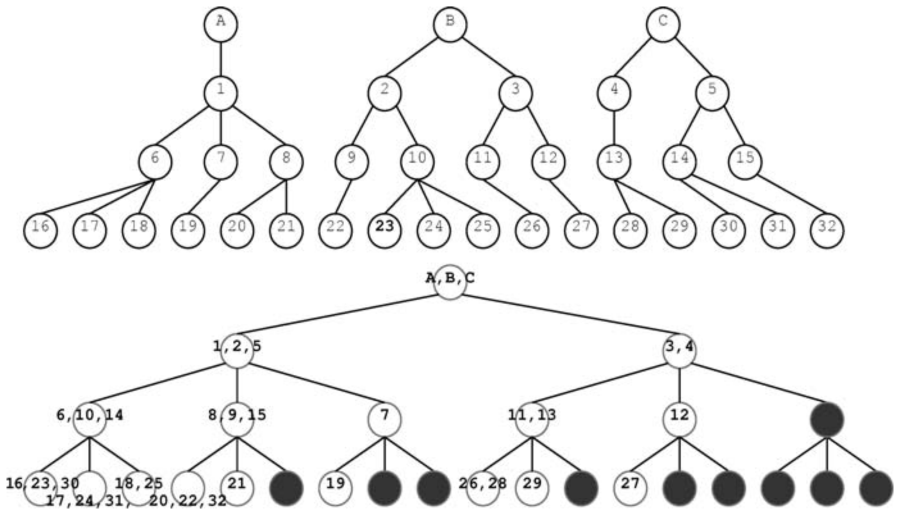


Fig. 6 Finding a supertree for trees A, B, and C using our simple heuristic

This guarantees that every tree $(V, E) \in F$ is embeddable into $(\bar{V}, \bar{E})$. This is illustrated by the entire bottom tree in figure 6.

- Children of a node $v \in \bar{V}$ have a strict ordering. In Figure 6 this strict ordering is left-to-right. In a computer implementation, it could be order within a list.

2. For each $(V, E) \in F$:

   (a) Assign the root v of (V, E) to the root $\bar{v}$ of the supertree $(\bar{V}, \bar{E})$.

   (b) Apply the $AssignTree$ operation below to the ordered pair $(v, \bar{v})$:

   $AssignTree(v_1, v_2)$:

   i. Sort the children of v_1 by degree, ascending. Break ties arbitrarily.

   ii. Assign the sorted children of v_1 to the children of v_2, *in order*. See the assignments in figure 6, especially the assignment of nodes 4 and 5.

   iii. Let $s(v)$ be the node of $(\bar{V}, \bar{E})$ to which v has been assigned. Apply *AssignTree* to each ordered pair $(u, s(u))$, where u is a child of v_1.

3. Delete any node in $(\bar{V}, \bar{E})$ which has no nodes assigned to it. Those are the nodes darkened out in Figure 6.

A pseudo-code is given in Algorithm 1. The ellipses in the code indicate that portions were removed for clarity.

Once the procedure is complete, we have a new frame for the aggregated stages, and an assignment we can use to generate the information for each new scenario correctly. The heuristic was able to generate an optimal supertree for the example in figure 6, but it is easy to show that it can be made to produce sub-optimal supertrees.

4.3. Generating and linking the subproblem deterministic equivalents

The next step is to generate and link the SDEs. Say that we are aggregating stages t_f through t_l. A (potentially) different SDE is generated for each node of the scenario tree at the first

Algorithm 1 Our Simple Heuristic for Supertrees.

```
class SuperTree(CompleteTree):
    ...
    def AssignTree(self, tree):
        tree.children.sort(SortByDegree)
        for i in range(0,len(tree.children)):
            self.children[i].AssignTree(tree.children[i])

    def PruneEmpty(self):
        ...

def GetSupertree(forest):
    # Assumes all trees complete with same depth
    degrees = []
    for d in range(1,forest[0].Depth()+1):
        maxDegree = max([t.MaxDegreeAtDepth(d) for t in forest]))
        degrees.append(maxDegree)
    super = SuperTree(degrees)
    for t in forest:
        super.AssignTree(t)
    super.PruneEmpty()
    return super
```

aggregated stage t_f. This SDE is generated not for any particular subtree, but for the supertree calculated at the previous step. In the *core* file, the information of the original *root* scenario is copied for each node of the supertree.

When generating the SDE, the elements in the $c_{t,\mathcal{A}}$ vector need to be multiplied by the *conditional* probability of each bundle $\mathcal{A} \in \Lambda_t$. This adjusts the coefficient values for the probabilities associated with the new scenario structure. When the solver processes the resulting file, the correct values (the unconditional probabilities) are restored.

If the last stage of the aggregation is not the last stage of the original problem, then the information for the stage immediately following it also needs to be amended. As explained in 4.1, stage t_l has many more variables than it had before, most of which we must not consider in the linkage to any individual scenario at stage $t_l + 1$. Thus, the T_{t_l+1} matrix needs to be updated for every scenario in stage $t_l + 1$, by adding zeroes to the appropriate columns. Remember that each T_{t+1} matrix refers to variables in stage t, but the matrix itself, and its associated constraints, are better thought of as belonging to stage $t + 1$.

5. Lagrangian Relaxation

When the individual scenario problems represent a generalized network, or can in some other way be solved by a specialized algorithm, it can be profitable to use the explicit (or split-variable) representation for the non-anticipativity(Mulvey and Vladimirou, 1992). Often, in these cases, solution methods based on lagrangian relaxation such as the one in Rockafellar and Wets (1991) are used.

Lagrangian relaxation is most often used inside branch and bound schemes to produce bounds which are often better than, and always at least as good as, those produced by linear programming relaxations. It can also be used as a general-purpose solution technique, although this is less common. In stochastic programming, there are examples of general methods based on lagrangian relaxation, such as (Carøe and Schulz, 1999), a branch and bound method for stochastic integer programming, and (Rockafellar and Wets, 1991), which solves problems decomposed by bundle and includes a penalty component in the objective, among other similarities to an augmented lagrangian approach. There are also examples (in electricity generation see (Takriti and Birge, 2000) and (Nowak and Römisch, 2000)) of the use of lagrangian relaxationas part of a problem-specific approach.

The L-shaped method, as well as methods that solve DEs directly, can be considered to be closed procedures. Given a correct input problem, they are expected to return the solution to the problem. In contrast, lagrangian relaxation is usually *part* of a solution strategy, rather than a solution methodology in itself. In addition to relaxing constraints and solving decomposed problems with penalties, other components need to be added in order to obtain a complete solver, such as a multiplier search function, or a branch and bound manager. These components are usually problem-specific.

Thus, our approach to lagrangian relaxationis to provide all the *non*-problem-specific components. We provide everything that can be deduced from the algebraic structure:

- The pricing problem 2.5, in one MPS file; or its decomposition by scenario 2.6, in several MPS files.
- An objective calculation function, in a high level programming language.

The pricing problem is the explicit DE like the one in Figure 3 (formulation 2.2) absent the non-anticipativity constraints, and with an amended objective. There are two options available in our code. In the first, the entire instance is written out to a single file, and figuring out that the problem is decomposable is left to the solver. In the second, our code actually generates $|S|$ different files. The main advantage of lagrangian relaxationis decomposing the DE into one problem for each scenario. This component takes care of writing the pricing problem in that way, if requested.

We also generate a procedure, written in a high-level language, that takes as a parameter a set of prices representing penalties for deviations between variables in different scenarios belonging to the same bundle, and returns the vector of objective coefficients incorporating the penalties.

There were different options for how the procedure that calculates the priced objectives should be implemented. The multipliers have different meanings depending on how the non-anticipativity being relaxed are written. In one option, as in Section 2, one of the scenarios in each bundle is labeled the reference scenario, and the non-anticipativityare written to forbid variables belonging to the other scenarios in the same bundle from acquiring different values. In another option, the non-anticipativity are written to forbid each scenario from being different from its next sibling (chosen arbitrarily from the same bundle). Yet a third option is to express non-anticipativity as a deviation from the expected values of each variable across the scenarios in the same bundle, as in Takriti and Birge (2000). We chose the first option because it seems more intuitive for the development of custom search procedures. It should be made clear that this procedure is only intended to be a sensible default. In some approaches a custom representation for the non-anticipativity is used, which would need to be accompanied by a new procedure for calculating the pricing objectives. Our system is written so that procedures reflecting different formulations of the non-anticipativitycan be readily added.

The aspects left for the modeler to provide are the problem-specific constructs:

- The search function.
- The final feasibility generator.

In any lagrangian relaxation-based approach, a procedure to search through the space of multipliers must be used. While there exists a general-purpose approach (the subgradient method (Fisher, 1981)) that can be efficient for some problems, it has not proved to be a good choice for general problems, because the step sizes needed to insure convergence absent special structure are too small to be practical. Thus this aspect of the solution methodology is left for the modeler to handle outside of our system.

Even if the pricing problem is solved to optimality, it is in general necessary to do extra work to obtain a solution to the original problem. The modeler would also need to provide a routine for that purpose.

6. The visual environment

The main objective of this research is to make the process of trying different solution methods for stochastic programming as painless as possible. Thus, the primary function of the Graphical User Interface (GUI) (see figure 7) is to aggregate the tools previously discussed and provide an interface to them that is intuitive, complete, and quick to learn and use.

A second function of the GUI is to provide information about the structure of the problem, and to allow easy navigation through different parts of the problem. Specifically, it provides information about the scenario tree, and displays the sparsity structure, or the matrix blocks themselves. The matrices are divided into the structures convenient for stochastic programming, as can be seen in figure 7. Notice the scenario tree on the left, and the sparsity structure on the right. The horizontal and vertical lines in the sparsity structure window separate the matrices W_t, A_t, T_{t+1}, W_{t+1}, the vectors c_t, c_{t+1}, h_t, b_t, and h_{t+1}, and bounds (if present). Elements that are stochastic are displayed in a different color, as are integer or binary elements.

All the transformations are available directly from the menus. Selecting some of them brings up dialog boxes designed for that specific transformation. The dialog box for the stage aggregation transformation can be seen in Figure 8. The modeler is given the option of

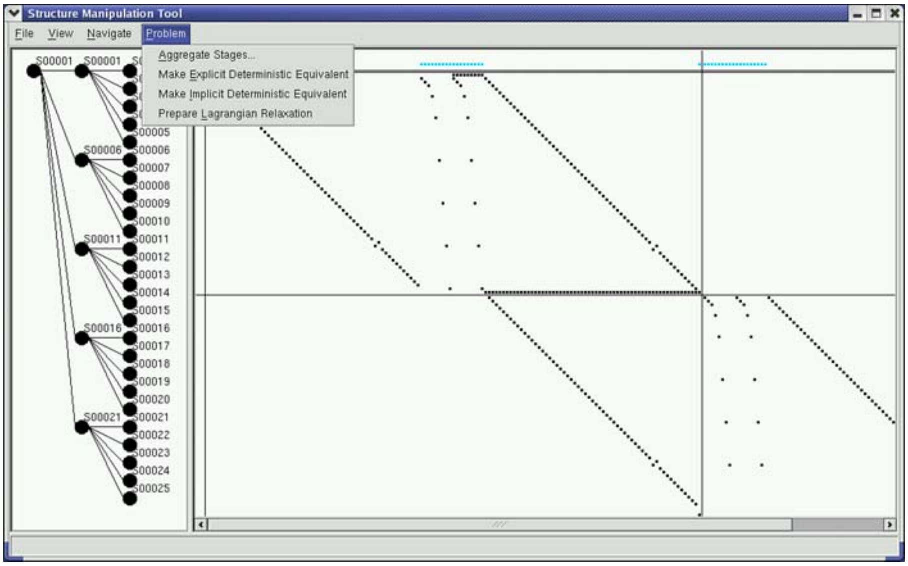
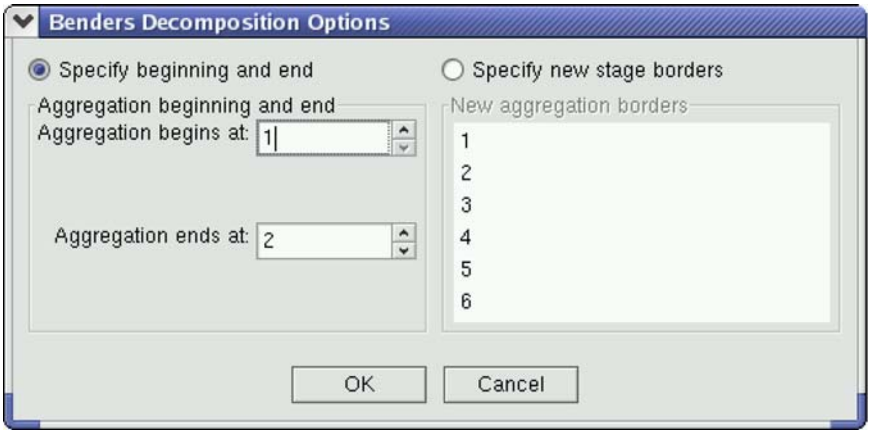


Fig. 7 The main window of the program


Benders Decomposition Options

Specify beginning and end [✓] / Specify new stage borders

Aggregation beginning and end
Aggregation begins at: 1
Aggregation ends at: 2

New aggregation borders
1
2
3
4
5
6

Fig. 8 The stage aggregation dialog

specifying a single range for the aggregation, or of specifying multiple new branch points as a subset of the original branch points.

From the GUI, the user can read SMPS files into the back-end database and write transformations from the database into MPS or SMPS files. Navigating through the scenario tree on the left window causes the data on the right window to be automatically updated. It is also possible to zoom in and out of the scenario structure, or to view the actual coefficients instead of simply the sparsity structure. Most importantly, the operations are all done quickly and effortlessly, saving modeler time.

As part of the research, we developed an extensive library for manipulating a database similar in semantics to that of an SMPS file. The library allows for accessing elements in terms of stages, matrices, scenarios, and other stochastic programming-specific concepts. For example, the library includes a method *GetTElems(stage,scen)*, which a solver developer could use to obtain the information about the technology matrix T_{ts}. The library has many such high-level procedures, so that algorithms can be written with constructs fairly close to the mathematical constructs we are used to dealing with.

The library is built on top of a relational database. This approach adds value in the debugging stage of a model. Imagine that the modeler suspects, for instance, that there is something wrong with some data item, or wishes to perform some aggregate analysis of the data. Even if there is no prepackaged method in the library that would support the specific investigation, because the data is in a relational database, the modeler can devise a SQL query to obtain the information desired.

Initially, the use of a relational database created efficiency problems, especially when the number of scenarios became large. These problems were solved by creating cached normalizations in temporary tables in memory. The cache, of course, is transparent to the solver developer using the library.

The data manipulation library's interfaces make no assumptions about the origins of the data. Thus, the library could be reimplemented to obtain data from another source, such as a modeling language interpreter or a file, without requiring any changes to applications built on top of it.

The GUI is modular, and designed to be extendible. As a consequence of this design, and of the library mentioned above, adding a module to perform some specific test or operation, such as a module for detecting network problems, can be done without having to learn major details about the GUI itself. To create a module, one would write a script that uses the manipulation library (or other functions needed by the particular module) to examine or manipulate the instances in the database. To make new functionality available from the GUI, only two simple actions are needed: adding an item to the desired point in the menu list; and associating the menu item with the script.

7. Conclusions and extensions

By bundling this set of tools into one system, we have been able to reduce the cost of developing stochastic programming based solutions, by allowing different solution methodologies and algebraic manipulations to be tried with very little effort, from a single representation. This allows designers of modeling languages for stochastic programming to create environments which are more problem-focused and less methodology-dependant. We have also been able to offer simple solutions to difficulties related to peculiarities of specific instances. For example, another research group sent us an instance of a very simple 3-stage problem that MSLiP (Gassmann, 1990) was failing to solve. After we aggregated stages 1 and 2 (or 2 and 3), we were able to solve that instance using the same solver.

Many extensions would be useful for this package. More solution approaches could be handled, such as regularized decomposition, augmented lagrangians, or sampling methods. It would also be useful to have more functionality in structure detection, as well as interfaces for manipulating solver parameters. In the handling of lagrangian relaxation, it would be interesting to allow for relaxation of constraints other than the non-anticipativity constraints. It would also be interesting to have code generators for other ways of writing non-anticipativity constraints.

Our system is easily portable to any modern computing platform. Its main components are: a database and related schemas; a library that makes accessing the schemas easier for stochastic programming applications; a series of scripts built on the database; and a GUI that hides the scripts, and provides a frame for graphical interaction and some basic functionality such as file handling. The system is written mostly in Python and MySQL.

We would like to acknowledge here the contribution given by the many developers and users of tools like Python, MySQL, SWIG, wxWindows, and others, without which the software development would have been much more difficult. We would also like to acknowledge the work of those involved with the NEOS server (Czyzyk, Mesnier, and Moré, 1998) and various solvers available on it (in particular MSLiP (Gassmann, 1990)), on which most of the testing was done.

Information about obtaining the system, including installation instructions and links to the software it uses, is available at `http://senna.iems.northwestern.edu/strums`.

Acknowledgements This research was supported in part by Grants DMI-9800077 and CCR-0082807 from the National Science Foundation. The authors would also like to thank the excellent comments and suggestions of two anonymous referees.

References

Benders, J.F. (1962). "Partitioning Procedures for Solving Mixed Variables Programming Problems." *Numerische Mathematik* 4, 238–252.

Birge, J., M. Dempster, H. Gassmann, E. Gunn, A. King, and S. Wallace. (1987). "A Standard Input Format for Multiperiod Stochastic Linear Programs." *COAL Newsletter* 17, 1–19.

Birge, J.R. (1985). "Decomposition and Partitioning Methods for Multi-Stage Stochastic Linear Programming." *Operations Research* 33, 989–1007.

Birge, J.R. and F. Louveaux. (1997) *Introduction to Stochastic Programming*. Springer-Verlag.

Carøe, C. and R. Schulz. (1999). "Dual Decomposition in Stochastic Integer Programming." *Operations Research Letters* 24, 37–45.

Condevaux-Lanloy, C. and E. Fragniere. (1998). Setstoch: A Tool for Multistage Stochastic Programming with Recourse. Technical Report, Logilab Department of Management Studies, University of Geneva, Switzerland.

Czyzyk, J., M.P. Mesnier, and J.J. Moré. (1998.) "The NEOS Server." *IEEE Computational Science and Engineering* 5(3), 68–75, July.

Dempster, M., J. Scott, and G. Thompson. (2002). Stochastic Modelling and Optimization using Stochastics. Technical Report, Judge Institute of Management Studies.

Dempster, M.A.H. and R.T. Thompson. (1998). "Parallelization and Aggregation of Nested Benders Decomposition." *Annals of Operations Research* 81, 163–187.

Fisher, M.L. (1981). "The lagrangian Relaxation Method for Solving Integer Programming Problems." *Managment Science* 27, 1–18.

Fisher, M.L. (1985). "An Applications Oriented Guide to Lagrangian Relaxation." *Interfaces* 15(2), 10–21.

Gassmann, H. (1990). MSLiP: A Computer Code for the Multistage Stochastic Linear Programming Problem. *Mathematical Programming* 47, 407–423.

Gassmann, H. and E. Schweitzer. (1996). "A Comprehensive Input Format for Stochastic Linear Programs." Working Paper, School of Business Administration, Dalhousie Uinversity, Halifax, Canada.

Goux, J.-P., S. Kulkarni, J. Linderoth, and M. Yoder. (2000). An Enabling Framework for Master-Worker Applications on the Computational Grid. "Technical report, Argonne National Laboratories."

Gupta, A. and N. Nishimura. (1998). "Finding Largest Subtrees and Smallest Supertrees." *Algorithmica* 21(2), 183–210.

Kall, P. and J. Mayer. (1996). Slp-ior: An Interactive Model Management System for Stochastic Linear Programs. *Mathematical Programming* 75, 221–240.

King, A. (1994). *SP/OSL V1.0, Stochastic Programming Interface Library, User's Guide*.

Klaassen, P. (1998). "Financial Asset-Pricing Theory and Stochastic Programming Models for Asset/liability Management: A Synthesis." *Management Science* 44, 31–48.

Linderoth, J. and S. Wright. (2001). "Decomposition Algorithms for Stochastic Programming on a Computational Grid." Technical Report ANL/MCS-P875-0401, Argonne National Laboratories, Apr.

Mulvey, J.M. and H. Vladimirou. (1992). "Stochastic network programming for Financial Planning Problems." *Management Science* 38(11), 1642–1664.

Nowak, M.P. and W. Römisch. (2000). "Stochastic Lagrangian Relaxation applied to Power Scheduling in a Hydro-Thermal System under Uncertainty." *Annals of Operations Research* 100, 251–272.

Rockafellar, R. and R.J.-B. Wets. (1991). "Scenarios and Policy Aggregation in Optimization Under Uncertainty." *Mathematics of Operations Research* 16, 119–147.

Slyke, R.V. and R.J.-B. Wets. (1969). "L-shaped Linear Programs with Application to Optimal Control and Stochastic Programming." *SIAM Journal on Applied Mathematics* 17, 638–663.

Takriti, S. and J. Birge. (2000). "Lagrangian Solution Tehniques and Bounds for Loosely Coupled Mixed-Integer Stochastic Programs." *Operations Research* 48(1), 91–98.

Valente, P., G. Mitra, C. Poojari, and T. Kyriakis. (2001). "Software Tools for Stochastic Programming: A Stochastic Programming Integrated Environment (SPInE)." Technical report, Brunel University, Uxbridge, UK UB8 3PH.

Van Slyke, R. and R.J.-B. Wets. (1967). "L-shaped Linear Programs with Application to Optimal Control and Sotchastic Programming." *SIAM Journal on Applied Mathematics* 17(4), 638–663.