Structure Exploiting Tool in Algebraic Modeling Languages*

Emmanuel Fragnière†, Jacek Gondzio‡
Robert Sarkissian, Jean-Philippe Vial

Logilab, Hec, Section of Management Studies, University of Geneva,
102 Bd. Carl Vogt, CH-1211 Genève 4, Switzerland

Logilab Technical Report 1997.15§
June 18, 1997, revised in June 28, 1998

Abstract

A new concept is proposed for linking algebraic modeling language and the structure exploiting solver. SPI (Structure Passing Interface) is a program that enables retrieving structure from the anonymous mathematical program built by the algebraic modeling language. SPI passes the special structure of the problem to a SES (Structure Exploiting Solver). An integration of SPI and SES leads to SET (Structure Exploiting Tool) and can be used with any algebraic modeling language.

Key words. Algebraic modeling language, large scale optimization, structure exploiting solver.

1 Introduction

Practitioners who use mathematical programming are confronted with a dilemma. On the one hand, their problems are usually so large and so complex that they cannot be modeled without the aid of an algebraic modeling language. On the other hand, large models often necessitate the use of a specialized structure exploiting solver. Unfortunately, algebraic modeling languages only access general purpose

*This research was supported by the Fonds National de la Recherche Scientifique Suisse, grants #12-42503.94 and #1214-049696.96/1.

†HEC, Department of Management, University of Lausanne, BFSH1, 1015 Dorigny-Lausanne, Switzerland

‡On leave from the Systems Research Institute, Polish Academy of Sciences, Newelska 6, 01-447 Warsaw, Poland

§URL: http://ecolu-info.unige.ch/~logilab

solvers; they have no provision for transmitting the structural information required by a specialized structure exploiting solver. Thus, practitioners are prevented from using the cutting edge optimization techniques, unless they undertake a major programming effort. The aim of this paper is to present a concept that bridges this gap.

Solving real life problems via mathematical programming is usually carried out through successive stages. First, the user builds a model that represents the problem under study, by means of a set of equations and/or inequalities among decision and state variables. Next, the equations are encoded in a specific language that allows an optimization solver to input the model and the data. Finally, depending on the form of mathematical programming problem, the user has to choose an appropriate optimization solver.

Since real life problems may involve thousands of constraints and variables, there is no way to generate by hand each constraint individually. Rather, the constraints are partitioned into classes, each class having its generic representative. The model builder just provides the explicit algebraic expression for the constraint representative. This step mimics the mathematical way model builders describe their problem in a clear, concise and efficient way. However, this formulation is not directly usable by optimization solvers. The purpose of algebraic modeling languages is to bridge the gap between the natural algebraic formulation of the problem by the user and the input data scheme for the solver.

In this setting, the user treats the solver more or less as a black box. This is rather safe, as long as the user deals with models of moderate size. Even large-scale problems can be solved quite efficiently by today's commercial or public domain solvers. However, to achieve more realism the users tend to build larger and larger models whose dimensions hit the time and storage limits of the more advanced combination of general purpose software and hardware. Indeed, real life problems often involve dimensions, such as time, uncertainty, physical location, that take multiple values. Each combination of these values across dimensions defines an equation. By compounding all possible values, a rather compact algebraic formulation may well give rise to an enormous model that is beyond the capabilities of the most advanced general purpose software.

The process of repeating with different values, each time with an appropriate set of variables, induces a model structure. If properly identified, the structure may be exploited by an appropriate optimization code. A well-known example is the block-angular structure of constraints that can be handled by a decomposition approach. Decomposition allows tight control of memory requirements and great computational efficiency, in particular through parallel computation. Of course, to implement these algorithms, it is mandatory to have an explicit description of the structure at hand.

In mathematical programming, a structure becomes apparent through an appropriate row and column ordering. In many instances, the user can analyze the situation beforehand and specify what kind of ordering will produce the desired structure. This information has to be conveyed from the algebraic formulation to the input file defining the mathematical program. Those languages follow some internal and systematic rules to generate the input data file. The resulting ordering of rows and columns is very unlikely to display any exploitable structure. This deters users from resorting to specialized optimization techniques, unless they are ready to invest heavily in computer programming. Therefore, there is a clear need to extend the concept of an algebraic modeling language to allow the transmission of information about the structure.

We propose such a feature. Our work is an outgrowth of Chang and Fragnière [7]. This paper was the first attempt at creating a general scheme for passing a model structure from the algebraic modeling to the solver. The authors were able to hook a decomposition scheme with the modeling language GAMS. They applied the technique to the solving of a nonlinear programming problem [29] arising from an economic application. They thus demonstrated the feasibility of the approach. There seems to be a rising interest in this concept, as it is shown by the subsequent paper of Entriken et al. [11].

Conceptually, a structure passing scheme is not more complicated than the model building itself. When writing down the model, the model builder can often specify classes or subsets of generic constraints and generic variables linked with one another. These subsets form two partitions, one for the set of variables and one for the set of constraints. From such partitions one can easily retrieve a block structure in a model, that might be exploited by a solver. We propose a standard for describing special structures of mathematical programming problems. The main point consists in defining a partition of the problem into blocks. The standard naturally covers most of the special structures that may be of interest to the solver, e.g., block-angular, staircase, row and column bordered, and many others, including nested ones.

The implementation of a structure exploiting device in connection with an algebraic modeling language raises an important issue. Namely, what should be the relationships between model building, structure passing and the solver? For instance, model building and structure passing could be both embedded in the algebraic modeling language. The implementation of this approach requires access and perfect knowledge of the programming of the language. Most likely, it would have to be performed by the designer of the language himself. Being language dependent, the task would have to be performed for each modeling language independently.

We propose an alternative approach that preserves the independence between the modeling lan-

guage and structure passing. Indeed, the structure passing interface can be entirely decoupled from the modeling language, just as the modeling language is separated from the solver. The map that allows the linking between structure passing, model building and the solver is the so-called dictionary. A dictionary is a data structure that is readily accessible to the user in at least two important modeling languages AMPL [14] and GAMS [6]. The dictionary gives the correspondence between the variables and constraints in the original algebraic model and those of the *anonymous* mathematical program sent to the solver. The structure passing interface uses this dictionary to reorganize variable and constraint indices so as to produce a block structured mathematical programming formulation.

In our current implementation of the structure passing interface, we are using a public domain software to automate this task. The task is reduced to the search of these words from the dictionary that match a given user specified keywords. Consequently, it reduces to the analysis of strings (regular expressions). To facilitate the implementation, we used the public domain regular expressions package, *Regex* [21], available through Free Software Foundation. We want to stress two points here. First, any other string analysis tool could be used instead of *Regex* to perform this task. Secondly, there are other ways of keeping track of the dimension inducing model structure in AML.

The paper is organized as follows. In Section 2, we give some examples of structures that are exploitable by specialized solvers. In Section 3, we explain why algebraic modeling languages loose structures and the concept SET we developed to retrieve and exploit them with adequate solution techniques. In Section 4, we describe in detail an implementation of the Structure Passing Interface (SPI) - a program that enables a connection between algebraic modeling languages and Structure Exploiting Solvers (SES). In Section 5, we present an implementation of one particular Structure Exploiting Tool (SET). This SET uses SPI to hook a specific SES (an interior point based decomposition solver) with the GAMS modeling language. We discuss some issues of this link to end with an example of its use. In Section 6, we give conclusions.

2 Exploiting structure in optimization algorithms

In this section we review some special structures of mathematical programs that can be exploited in the design of efficient optimization software. To simplify our presentation, we shall concentrate uniquely on linear constraints, but our considerations apply to nonlinear constraints, i.e., to the Jacobian matrix.

Let us start with two structures that are currently handled by most linear programming codes: the *network* problems and to the *generalized upper bounded* (GUB) problems [8]. If present in a problem, those structures allow the use of efficient algorithms that tremendously improve upon a

general purpose linear programming code. Fortunately enough, those structures are relatively easy to identify. Today's advanced implementations of the simplex method are able to detect automatically whether an LP contains an embedded network or a GUB structure.

The other structures we shall discuss are characterized by a special layout of the Jacobian matrix where nonzero elements appear in submatrices, or blocks. A first example is the so-called *staircase* structure which is often displayed by mathematical programs that model dynamic processes. The simplex method can take advantage of it, both in the routines that handle the basis inverse and in the pricing [13]. This structure is also advantageous for the interior point solver: Cholesky factors of AA^T matrices preserve the staircase form and display moderate fill-in. Moreover, there exists an ordering that preserves the block structure and yields a product matrix AA^T suitable to parallel Cholesky decomposition [25].

Block-angular structures make an LP problem eligible for decomposition. The *primal block-angular* structure can loosely be described as the juxtaposition of two sets of constraints: the complicating constraints and the easy ones. Without the complicating constraints, the problem would be separable and therefore easy to solve. The standard structure exploiting solver for this class of problems uses a Lagrangian relaxation of the complicating constraints to split the problem into subproblems.

In the linear case, this amounts to the well-known Dantzig-Wolfe decomposition [9]. The *dual block-angular* structure contains a set of linking variables. Should these variables be eliminated, the LP would become separable. This structure can be exploited via Benders partitioning of variables [3], the dual counterpart of Dantzig-Wolfe decomposition.

Superposing primal and dual block-angular structures leads to the *block row and column bordered* matrices. Those matrices can be rearranged to display structures eligible to decomposition [12]. An alternative way to exploit this structure is to use a special row and column ordering in the building of the Cholesky factors of the AA^T matrix [27].

Finally, let us mention the whole variety of more complicated structures that, in particular, can be the result of nested embedding of some of the basic structures described earlier. A classical example of such a structure arises in multi-stage stochastic programming problems [4].

Despite their variety, the above-mentioned structures share a common property: all of them can be described by a block-partitioning of the constraint matrix.

Let $A \in \mathcal{R}^{m \times n}$ be the block partition representation of the constraint matrix

$$A = \begin{bmatrix} A_{00} & A_{01} & \dots & A_{0N} \\ A_{10} & A_{11} & \dots & A_{1N} \\ \dots & \dots & \dots & \dots \\ A_{M0} & A_{M1} & \dots & A_{MN} \end{bmatrix}. \tag{1}$$

The elements $A_{kl} \in \mathcal{R}^{m_k \times n_l}$, for $k = 0, 1, \dots, M$ and $l = 0, 1, \dots, N$, satisfy $\sum_{k=0}^{N} m_k = m$ and $\sum_{l=0}^{N} n_l = n$. Below, we show how some well-known structures can be reflected in such a block partitioning representation of A.

1. Primal block-angular structure [9], see Figure 1. This amounts to $A_{kl} = 0$, $\forall k = 1, 2, \dots, M$ and $\forall l \neq k$. $(M = N)$.

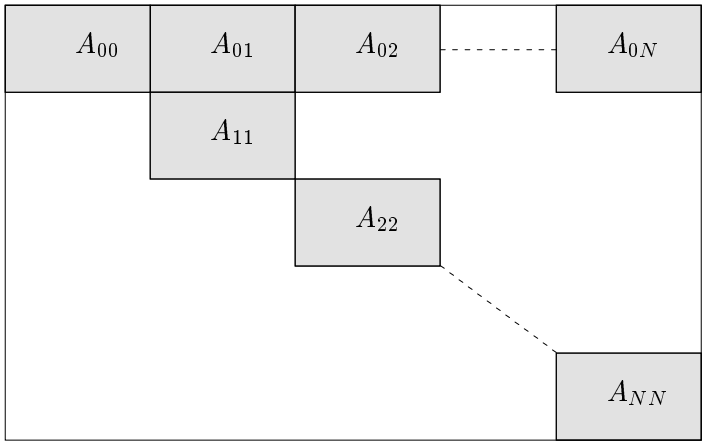


Figure 1: Primal block-angular structure.

2. Dual block-angular structure [3], see Figure 2. This amounts to $A_{kl} = 0$, $\forall l = 1, 2, \dots, N$ and $\forall k \neq l$. $(M = N)$.

3. Staircase structure [13, 25], see Figure 3. This amounts to $A_{kl} = 0$, $\forall k, l$ such that $k \neq l$ and $k + 1 \neq l$. $(M = N)$.

4. Block row and column bordered structure [12, 23], see Figure 4. This amounts to $A_{kl} = 0$, $\forall k = 1, 2, \dots, M$ and $\forall l \neq k$, $l \neq 0$. $(M = N)$.

The formal presentation of a set of constraints is not unique. Indeed, a permutation in the variable and in the constraints orderings, produces an equivalent mathematical program. To be more precise,

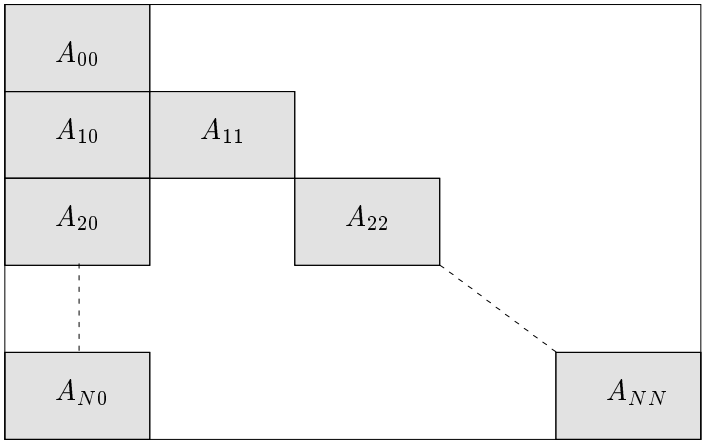


Figure 2: Dual block-angular structure.

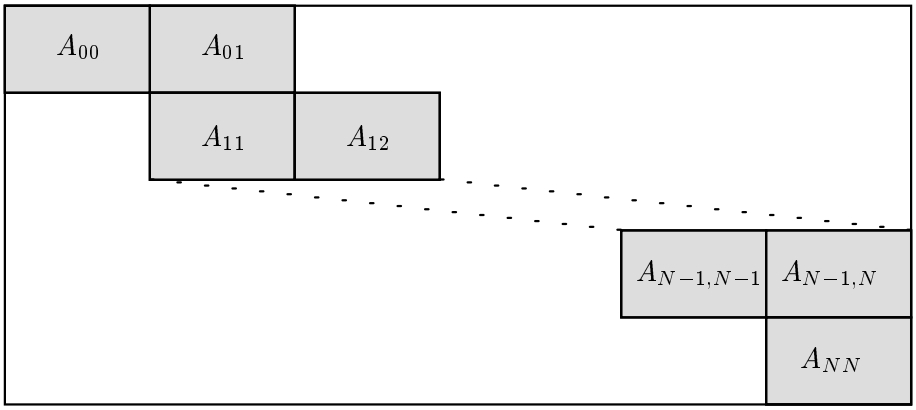


Figure 3: Staircase structure.

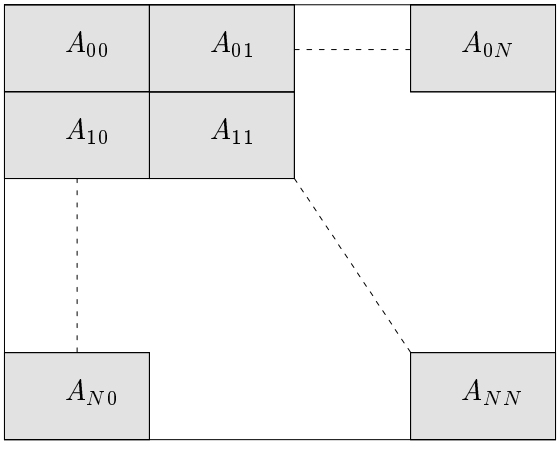


Figure 4: Block row and column bordered structure.

let $P \in \mathcal{R}^{m\times m}$ and $Q \in \mathcal{R}^{n\times n}$ be two permutation matrices. Define $\tilde{A} = PAQ$, $\tilde{b} = Pb$ and $\tilde{c} = Q^T c$. The linear program

$$\min\left\{\tilde{c}^T\tilde{x} \mid \tilde{A}\tilde{x} = \tilde{b},\ \tilde{x} \geq 0\right\}$$

is equivalent to

$$\min\left\{c^T x \mid Ax = b,\ x \geq 0\right\}$$

for $x = Q\tilde{x}$.

As we shall discuss in the next section, an algebraic modeling language presents permutations that almost surely produce a Jacobian matrix A with no apparent structure. Our main concern is to construct new permutations that will disclose the structure and allow appropriate algorithms to exploit it. If we ignore the permutations within the blocks A_{kl} of (1), then the information needed to transform $\tilde{A}$ with no block structure into A, with a revealed block structure, may be further simplified. In fact, we only need to specify the row and column partitions of A. Let $\mathcal{R}_k$, $k = 0, 1, \ldots, M$, and $\mathcal{C}_l$, $l = 0, 1, \ldots, N$, denote row and column partitions, respectively such that

$$\bigcup_{k=0}^{M}\mathcal{R}_k = \{1, 2, \ldots, m\} \text{ and } \mathcal{R}_{k_1} \cap \mathcal{R}_{k_2} = \emptyset,\ \forall k_1, k_2 \in \{0, 1, \ldots, M\},\ k_1 \neq k_2 \tag{2}$$

and

$$\bigcup_{l=0}^{N}\mathcal{C}_l = \{1, 2, \ldots, n\} \text{ and } \mathcal{C}_{l_1} \cap \mathcal{C}_{l_2} = \emptyset,\ \forall l_1, l_2 \in \{0, 1, \ldots, N\},\ l_1 \neq l_2. \tag{3}$$

The above conditions mean that each row $i = 1, 2, \ldots, m$ of $\tilde{A}$ belongs to only one set $\mathcal{R}_k$ and each column $j = 1, 2, \ldots, n$ of $\tilde{A}$ belongs to only one set $\mathcal{C}_l$. Hence the definition of partitioning (2-3) reduces to determining two mappings for the rows and columns of $\tilde{A}$, respectively. Let the mappings be

$$M_{\mathcal{R}} : \{1, 2, \ldots, m\} \longrightarrow \{0, 1, \ldots, M\} \quad M_{\mathcal{R}}(i) = k \quad \text{iff} \quad i \in \mathcal{R}_k \tag{4}$$

and

$$M_{\mathcal{C}} : \{1, 2, \ldots, n\} \longrightarrow \{0, 1, \ldots, N\} \quad M_{\mathcal{C}}(j) = l \quad \text{iff} \quad j \in \mathcal{C}_l, \tag{5}$$

respectively.

These mappings bring the minimum information to transform matrix $\tilde{A}$ into A in (1). The mappings do not specify the permutations of rows and columns within each block A_{kl}. Consequently, there exist many permutations P and Q that produce an appropriate block-structured matrix A.

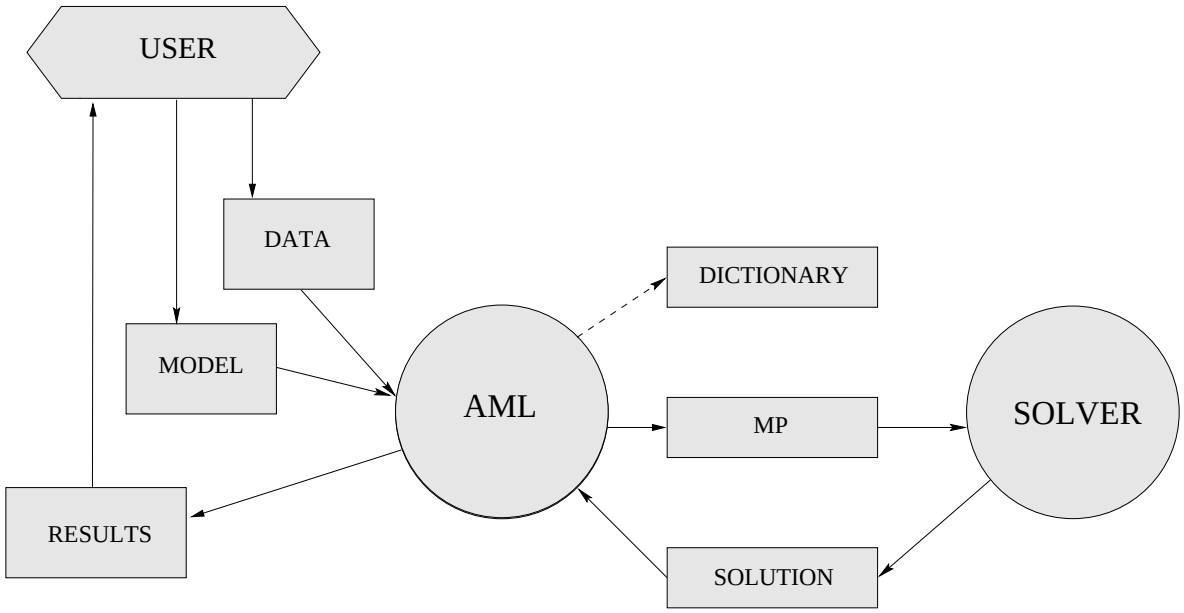


Figure 5: Links between an AML and a solver.

3 Loss of structures in algebraic modeling languages

An algebraic modeling language (AML) is the bridge between the algebraic formulation provided by the user and the mathematical programming formulation fed to the solver. Figure 5 shows its classical usage. The user provides the AML with the model written in equivalent algebraic formulation and its data. The AML then generates the complete mathematical program (MP) and sends it to the solver. At that point, all operations are automated. Once the optimization is completed, the solver sends back a solution to the AML. The user finally extracts from AML tables of results.

In Section 3.1, we explain why algebraic modeling languages loose any structure that could be easily detected in the algebraic formulation. In Section 3.2, we explain the SET (Structure Exploiting Tool) concept, a solution we developed to retrieve structures in algebraic modeling languages to be able to exploit them with appropriate solution techniques.

3.1 Why the structure is lost?

There are many ways of translating the algebraic formulation into the mathematical program. Indeed, every AML uses its own ordering scheme to generate the problem and clearly, there are many ways to do this in practice. The different mathematical programs that can be produced from the same model are equivalent, modulo two permutations of the Jacobian matrix, one for the rows and one for the columns.

Practical models usually define a number of different collections of variables: x_{ij}, y_{ij},z_{ij} and so on. The most likely way to process these variables in the algebraic modeling language would be to treat only one collection at a time, with the result that each collection appears separately: first all the x_{ij} variables, then all the y_{ij} one, and so forth. Such an ordering cannot possibly correspond to the order of the blocks in the model's structure, since in general there are some variables from every collection in every block. The same problem applies to the generation of the constraints.

In the usual link between AML and the solver, this ordering remains an internal data structure of AML and is never passed to the solver. The solver works with the *anonymous* formulation of the mathematical program without any reference to the algebraic formulation. So the solver just ignores the ordering. The AML has to store it: once the optimal solution is returned from the solver, AML will reuse this ordering to express the solution in terms understandable to the modelbuilder, i.e., in terms of constraint and variable names in the original algebraic formulation.

In some cases the information is stored implicitly in AML and is not directly accessible to the user. However, in most cases, this information is stored explicitly in a form of a *dictionary* that can be accessed by the user or by a developer. AML uses this dictionary to process the output of the solver and present it in a user readable format. Naturally, such a dictionary can be organized in many different ways: every AML uses its own scheme to store the correspondence between the constraints and variables of the algebraic model and those of the mathematical program. However, AML usually builds its dictionary from the names of generic constraints or variables and from the elements of their attached sets. For example, with a generated variable x_{1A}, we could associate a string `x.1.A`[1] as a result of the concatenation of several substrings separated by specific separators.

Let us analyze in more detail ways to construct the one-to-one mapping that translates the user's algebraic formulation of the problem into a complete mathematical program. To this end, we consider as an example the MERGE model developed by Manne and Richels [26]. MERGE is a convex nonlinear programming problem written in GAMS. It is a model for evaluating the regional and global effects of Greenhouse Gases reduction policies. It quantifies alternative ways of thinking about climate change. The model may explore views on a wide range of contentious issues: costs of abatement, damages of climate change, valuation and discounting. Figure 6 displays the Jacobian matrix produced by GAMS for a reduced version of MERGE. GAMS also creates a row and column dictionary as shown in Figure 7.

Two categories of names are usually stored for each row and column. The first category consists

[1]The string corresponding to x_{1A} in the dictionary of AMPL modeling language [14], for example, has the form `x[1,'A']` and in the dictionary of GAMS modeling language [6] has the form `x(1,A)`.

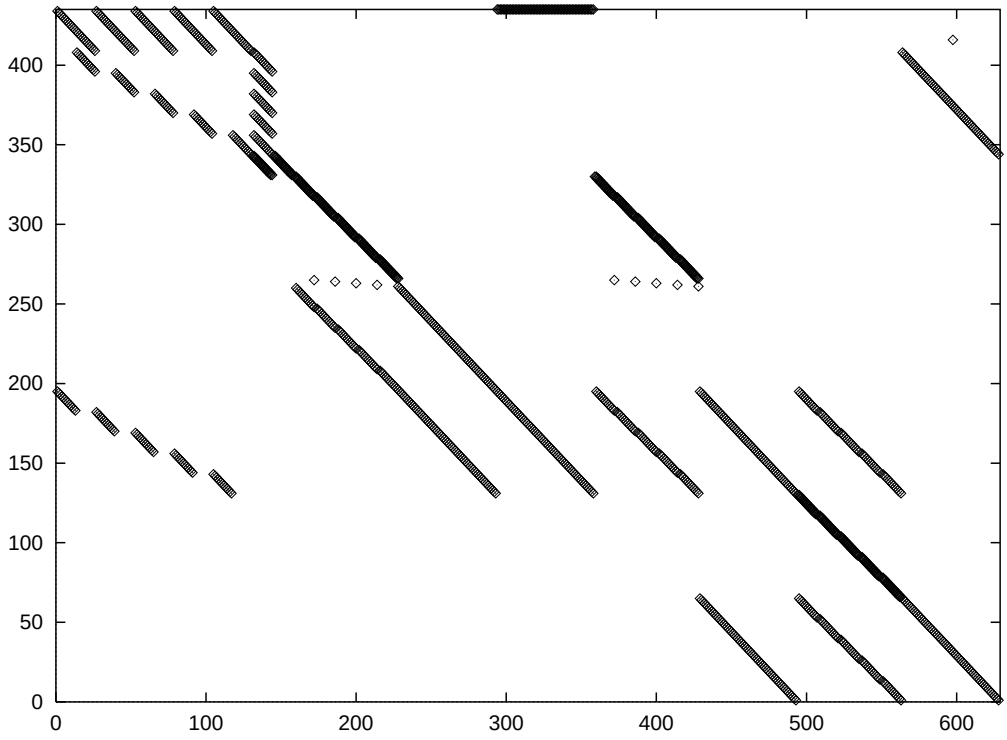


Figure 6: Anonymous matrix generated by GAMS for the MERGE model.

```
Extract of the GAMS row dictionary:
CAPACCUM(ROW,2100)
CAPACCUM(ROW,2110)
CAPACCUM(ROW,2120)
TC(USA,2120)
TC(OOECD,2120)
TC(USSR,2120)
TC(CHINA,2120)
TC(ROW,2120)
PRODT(USA,2000)
PRODT(USA,2010)
PRODT(USA,2020)
PRODT(USA,2030)
Extract of the GAMS column dictionary:
X(CHINA,CRT,2080)
X(CHINA,CRT,2090)
X(CHINA,CRT,2100)
X(CHINA,CRT,2110)
X(CHINA,CRT,2120)
X(ROW,NUM,2000)
X(ROW,NUM,2010)
X(ROW,NUM,2020)
X(ROW,NUM,2030)
X(ROW,NUM,2060)
```

Figure 7: Extract of the dictionary produced by GAMS.

of the original name of the generic equation or the generic variable that has generated a specific row or column. The second category is a list of elements that are generated for each index associated with the corresponding equation or variable. Let us now observe that all relevant information on constraints and variables that was contained in the algebraic model is now stored in the dictionary (names of constraints/variables, elements of sets). With the help of this dictionary, an external user could identify the rows and columns of the anonymous mathematical program with their equivalents in the algebraic formulation (this analysis is in fact done within the AML once it receives the solution from the solver).

3.2 The concept of structure exploiting tool

To solve a large structured mathematical program, the user must have some idea of what constitutes an exploitable structure, and what kind of solution method could take advantage of it. This requires of course some degree of sophistication from the user, though, safely enough, we may assume that advanced users are able to see whether their problem has, e.g., a staircase structure or a block-angular structure. The information about the structure can be expressed in relatively simple terms, usually by means of a serie of indices running over sets of locations, materials, time periods or scenarios in the problem. Translating this information in data exploitable by an appropriate solver is a much more demanding task that the user does not want to perform.

At present, there is no practical way in the AML to elicit this information. In this section, we analyzed the way AMLs function. We concluded that whatever the way the user writes the model, the AML will almost surely hide the structure when formulating the mathematical program passed to the solver.

The tool we propose is composed of two independent components: a module to pass the structure information from the user to the solver; and a module to exploit the structure in the optimization process. The acronym for the first module is SPI: *Structure Passing Interface*. We name the other module SES: *Structure Exploiting Solver*. SPI and SES together give SET: *Structure Exploiting Tool*.

We want to stress that SET is an extension of an AML, much more than a substitute for it. The integration of SET within AML does not necessitate any change in the usual application of AML. To exploit the problem structure, the user has to provide structural information to SPI, as can be seen in Figure 8. SPI translates this additional information into the description of the particular structure of the MP. It uses the AML dictionary to interpret the user supplied information and to translate it into the appropriate partitioning of the model.

We emphasize that SPI does not change the model. Thus, the communication between the AML

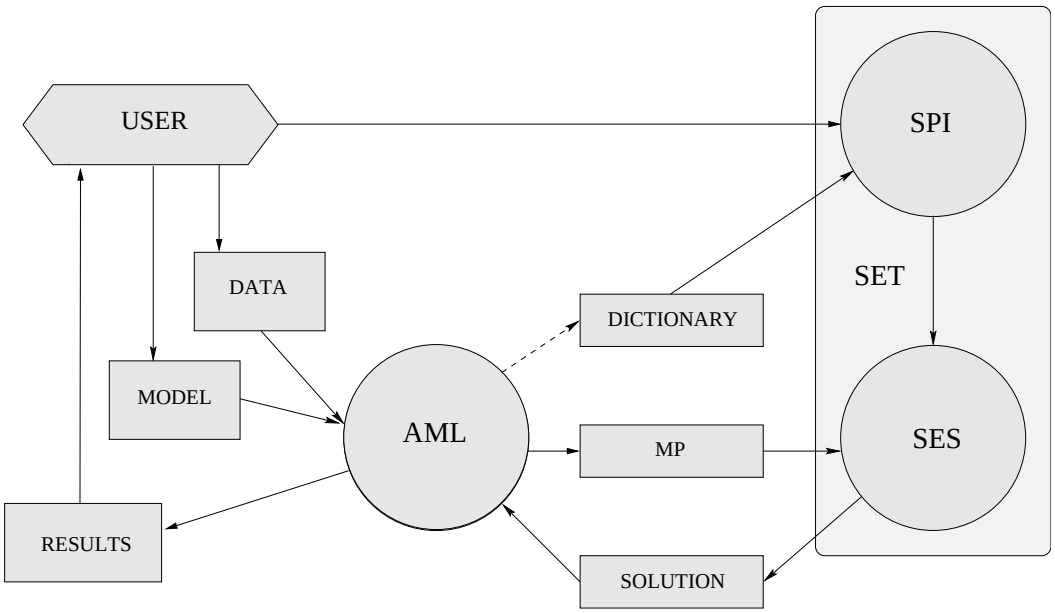


Figure 8: Integrating SET with an AML.

and the solver is not altered. SPI sends the relevant partitioning of the mathematical program directly to SES, the structure exploiting solver.

To sum up, SPI needs a bit of information from the user, plus the dictionary of the AML. From this information, SPI produces the partitioning that retrieves the special structure from the mathematical program. This partitioning is sent to SES (cf. Figure 8).

4 An implementation of SPI

Most of the structures that could be exploited by the solver are usually a consequence of the use of sets in the algebraic formulation of the model. Set indexing in the algebraic model replicates entire blocks of constraints and variables in the model for every element from the sets. The strings in the dictionary preserve this indexing and, in consequence, contain a full description of the structure.

In Section 4.1, we show how this particular implementation of SPI retrieves the structure by performing some pattern mathing on the dictionary. In Section 4.2, we present an example where SPI is used two retrieve two different structures from a model written in GAMS.

4.1 Retrieving the structure from the dictionary

SPI is a tool that uses the dictionary to retrieve the structure. Its basic operation consists in finding appropriate indices in the strings in the dictionary and attributing with every row and column of MP

the indices of partitioning to which they belong. SPI helps retrieve the structure but it is not an automated tool. It is the responsibility of the user to specify the desired structure. The user indicates an algebraic set that induces the structure he wants to retrieve in the mathematical program. Note that by the choice of different algebraic sets, the user can retrieve different interesting structures from the same problem. We addressed this issue in more detail in a subsequent paper [15].

In other words, SPI receives from the user the patterns specifying the row and column partitioning one wishes to preserve in the mathematical program. There are M patterns that identify row partitioning and N patterns that identify column partitioning. After having read the AML's dictionary of row names, it scans every word in it to find which row patterns it matches. If the pattern l matches row name i, then $M_{\mathcal{R}}(i) = l$. The same process is executed for column names of AML's dictionary and column patterns specified by the user. This analysis ends up with defining row and column mappings $M_{\mathcal{R}}$ and $M_{\mathcal{C}}$, (see equations (4) and (5), respectively).

Without going into more technical details, SPI analyzes every word from the row (or column) dictionary and checks whether it contains user specified substrings. Different substrings determine different partitions to which a given row (or column) belongs. To perform its task, SPI employs the notion of *regular expression* and a public domain `Regex` library [21]. By *regular expression* (or *pattern*), we mean a string that describes some set of strings according to some defined rules. We say that a regular expression r *matches* a string s if the latter belongs to the set described by r. Regular expressions are commonly used in computer science. For instance, many Unix [22] programs such as shells or editors have all their own dialect of regular expressions. Every user of Unix has certainly applied a regular expression, possibly without being aware of it.

Consider for instance the regular expression "BAL19[56][02468]\w". Its pattern contains two *bracket expressions*, which are "[56]" and "[02468]". Each bracket expression matches any single character that it contains. The expression "\w" matches any word. This regular expression thus matches all strings of the form (BAL)(date)(word) where 'date' is any even date between 1950 and 1968 and 'word' is any possible word. Regular expressions offer many more possibilities; but, their description is out of the scope of our presentation. A good implementation of regular expressions is available through the Free Software Foundation. The program, *Regex* [21], allows one to determine, among other things, whether a string matches a given pattern.

We should mention that more complicated patterns can require the use of more complicated keywords. Note for example, that to produce dates divisible by three one would have to use logical `OR` operator of *Regex*. We have met this case in practice when analysing very complicated models. Let us also observe that modeler can render his model easier for the use of structure exploiting tool.

To understand how this can be done let us remind that the modeler decides on the following three elements when writing his model:

- the generic name of the constraint or variable;
- the sets of elements;
- the order of appearance of index sets.

To end this section we would like to stress that whenever the model contained a dimension that induced a particular structure and that it was appropriately translated within AML, we were always able to retrieve the structural information from the dictionary. Clearly, *Regex* [21] (regular expression package) is only a tool that enabled us to automate this task. First, we could have used any other string analysis tool instead of *Regex*. Secondly, there are other ways of keeping track of the dimension inducing model structures in AML. For instance, in object-oriented modeling languages [2, 10, 17, 28, 31] retrieving structure would likely not necessitate the use of a dictionary.

4.2 An application of SPI with GAMS

Connecting SPI to GAMS consists in reading GAMS dictionary of row and column names. This is done by means of the GAMS library of Input/Output routines [16]. These routines can be called either by FORTRAN or by C programs. The library includes, in particular, a collection of routines that retrieve GAMS dictionary of row and column names associated with a given model.

SPI reads the GAMS row and column dictionary and then the user supplied regular expressions that specify the partitioning of the model. Next, SPI analyzes every row and column name from the dictionary to determine the appropriate regular expression that matches it. The analysis ends with a definition of row and column mappings $M_{\mathcal{R}}$ and $M_{\mathcal{C}}$, respectively.

To "understand" GAMS dictionary, SPI reads an appropriate GAMS-dedicated partitioning file. By default, if the algebraic model is supplied to GAMS in the `model.gms` file, the partitioning data is read from the file `model.set`. The current implementation of SPI employs *Regex* and reads partition information from the file called SET file. Naturally, linking SPI with a different modeling language would require specifying a different partition file; its regular expressions will have to be consistent with a different row and column dictionary.

Returning to the MERGE model, we can show how SPI is used, linked with GAMS, to retrieve two different block structures. The decomposable structure can be produced by taking into account the five regional blocks. First, we present the SET file in Figure 9 that was written to obtain a

Dantzig-Wolfe structure from this MERGE model. The keywords specified in this file use *Regex* [21] commands. Again, we made this choice to facilitate the implementation of SPI. Conceivably one could further simplify the syntax of information specified in `model.set` file and get it closer to the modeling language. NBSUBPB indicates that the Dantzig-Wolfe structure contains 5 subproblems. The five regions are: USA, OOECD, USSR, CHINA and ROW. Pattern matchings are performed for the rows as well as for the columns. Note that to specify a primal block-angular structure, it would have been sufficient to provide SPI with the row partition (the first five lines of the `model.set` file). The master problem is formed by the generic variables TRDBAL and CUME (see ROWMASTER). The generic variables ZG and ZC appear only in the master problem (see COLMASTER). The name of the objective function is NWELDF and the name of the objective variable is NWEL (for more information, a limited version of MERGE written in GAMS can be directly downloaded from the URL: `http://www-leland.stanford.edu/group/MERGE`).

```
NBSUBPB 5
# Selecting the rows
ROWSPB \(\w*(USA,\w*)\)
ROWSPB \(\w*(OOECD,\w*)\)
ROWSPB \(\w*(USSR,\w*)\)
ROWSPB \(\w*(CHINA,\w*)\)
ROWSPB \(\w*(ROW,\w*)\)
# Selecting the columns
COLSPB \w(USA\(.\)*)\|\(\w*(USA,\w*)\)
COLSPB \w(OOECD\(.\)*)\|\(\w*(OOECD,\w*)\)
COLSPB \w(USSR\(.\)*)\|\(\w*(USSR,\w*)\)
COLSPB \w(CHINA\(.\)*)\|\(\w*(CHINA,\w*)\)
COLSPB \w(ROW\(.\)*)\|\(\w*(ROW,\w*)\)
OBJROW NWELDF
OBJCOL NWEL
ROWMASTER \(TRDBAL(\w*,\w*)\)\|\(CUME(\w*)\)
COLMASTER \(ZG(\w*)\)\|\(ZC(\w*)\)
WITHGNUPLOT
DW
```

Figure 9: An example of a SET file to extract a Dantzig-Wolfe structure from MERGE.

From this SET file, SPI is able to split appropriately the mathematical program. Figure 10 presents the reordered Dantzig-Wolfe matrix generated by SPI. Modifying slightly the SET file we are also able to produce the row and column bordered structure as shown in Figure 11.

5 Accessing a particular SES from the modeling language

SES is part of SET which is problem dependent. For instance, if the user deals with a stochastic programming problem, the privileged structure is dual-block angular. Then, one may want to use a Benders like decomposition scheme. Moreover, the solver may work serially, or in parallel. The

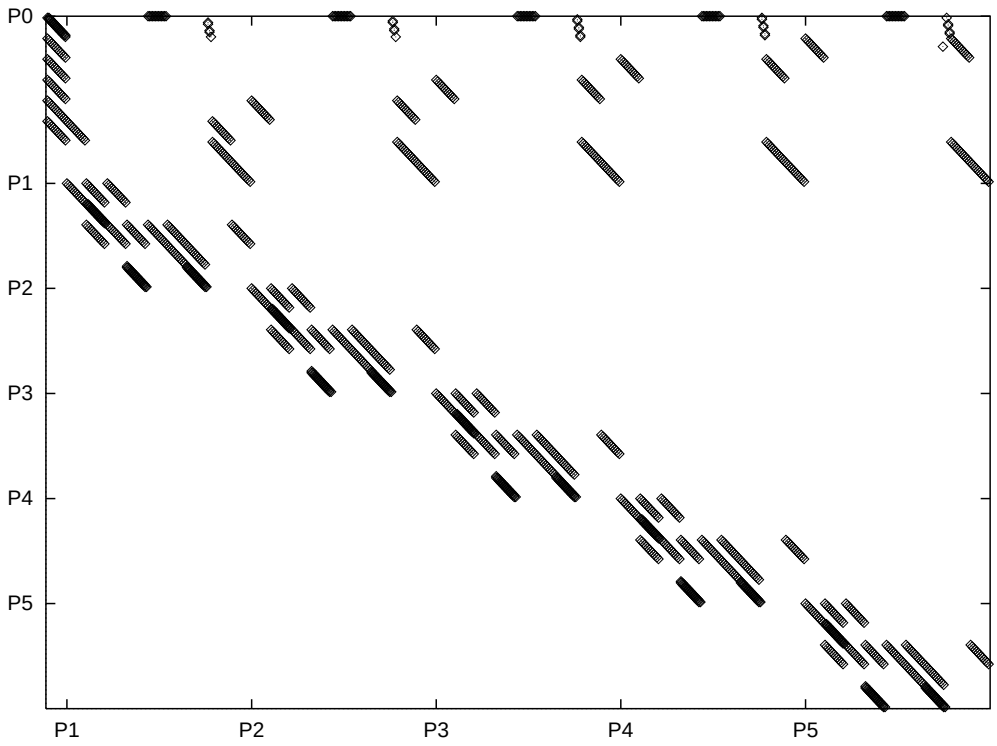


Figure 10: Dantzig-Wolfe decomposable structure extracted from MERGE.

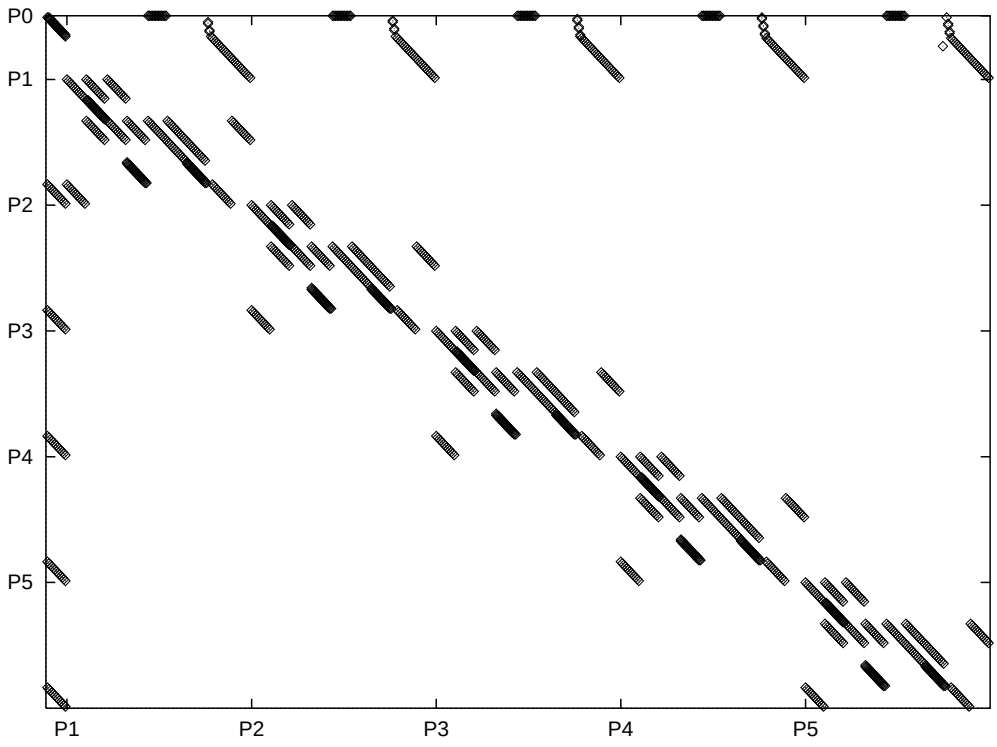


Figure 11: Row and column bordered structure extracted from MERGE.

same structure can also be exploited by a specialized interior point algorithm, see e.g., [5]. Therefore, there is a sharp difference in the respective roles of SPI and SES. Although the two components are intimately linked in any application of SET, SPI is the same for all SES, while SES has to be adapted to the problem at hand. Our goal is just to make sure that the structural information delivered by SPI is identifiable and usable by any SES.

In order not to prejudge a specific application, we do not describe our structure exploiting solver. In Section 5.1, we shall present an application that uses an interior point based decomposition. Even there, we shall not describe the solver, but we shall rather detail the way SET integrates an AML, SPI and an SES. In Section 5.2, we will present a complete application of SET where an energy model is generated by GAMS and is solved by an interior point decomposition.

5.1 An interior point decomposition hooked to GAMS

We have applied the ideas presented in this paper to connect an interior point based decomposition [20] with the GAMS modeling language [6]. More specifically, SPI is used to retrieve a primal block-angular structure of a problem and to pass it to a Dantzig-Wolfe type decomposition code.

The choice of the model structure, modeling language and the solver is not exclusive. Our purpose is just to illustrate how one can implement SET in practice. We thus offer a toolbox that will enable the users to hook their AMLs of choice to cutting edge solvers that exploit the user specified structure of the problem. The example belongs to a particularly relevant class in practice. Indeed, many large LPs have spatial, temporal, or stochastic dimension that generate structures amenable to decomposition. Though it is the case that a frontal approach, either the simplex or an interior point method, is often superior whenever the problem size remains compatible with the computational facilities, decomposition becomes an attractive alternative, if not the only one, when it is not compatible. Here, our aim is not to demonstrate the merits of decomposition. Indeed, we apply it to a moderate size example which could be solved more efficiently either by the simplex or by an interior point method.

In principle, interfacing SPI with a SES (cf. Figure 8) amounts to sending the mappings $M_{\mathcal{R}}$ and $M_{\mathcal{C}}$ from SPI to the solver. The solver receives, as usual, an anonymous mathematical program. This time, however, it receives also the partitioning information that makes it possible to transform the anonymous constraint matrix to the specially structured one. This step is done inside SES and it does not require any additional interaction with SPI.

The solver builds the permutations P and Q (as defined in Section 2) that transform the Jacobian of the anonymous mathematical program $\tilde{A}$ into its specially structured form A. Such permutations are easily obtained from the mappings $M_{\mathcal{R}}$ and $M_{\mathcal{C}}$. In fact, there are many permutations that satisfy

our needs: the solver is not concerned with the internal permutations of rows and columns within block A_{kl} of (1). After this step, matrix A already displays the special structure.

In the case of our decomposition code hooked to GAMS, matrix A has a primal block-angular structure as displayed in Figure 1. Hence, A is split immediately into blocks A_i, B_i, $i = 0, 1, \ldots, p$. Once the decomposition code finds the optimal solution, it is naturally expressed in terms of the original formulation. Now the permutations P and Q are used again to transform the optimal solution back to the ordering of the anonymous MP. In such form the solution is sent back to the AML.

In the case of the decomposition algorithm, we need to know which rows and columns of the complete LP belong to which block of the matrix. This can be characterized with very little information, since no reordering inside each block is necessary. This information is easily obtained from the algebraic formulation, especially in that particular case from information given by elements that form sets indexing of the algebraic formulation. For example, in a multiperiod model each variable and constraint is generated for a certain time period will belong to a subproblem. Every constraint that is linked to several periods belongs to the master problem (i.e., primal-block angular structure).

Many of today's large MPs generated with AMLs are decomposable. Due to their expected huge size they should be able to take full advantage of modern optimization techniques — interior point methods [1, 30]. Therefore, as a pilot implementation that uses SET, we have chosen the one based on interior point methods [20]. Moreover, this implementation employs two optimization codes freely available for research use: the Analytic Center Cutting Plane Method (ACCPM) - interior point based decomposition algorithm [19] and the Higher Order Primal-Dual Method (HOPDM) - general purpose interior point LP solver [18]. Both codes can be retrieved from the Web site `http://ecolu-info.unige.ch/~logilab/software/`.

5.2 An energy model generated by GAMS and solved by decomposition

The following application shows the use of a standard energy systems model called MARKAL [24] (MARket ALlocation). It has been developed under the aegis of the International Energy Agency and is currently implemented in more than 16 countries. MARKAL is a multi-period linear programming model. The objective function in the LP is the discounted sum, over the horizon considered (usually between 30 and 45 years), of investment, operating and maintenance costs of all technologies, plus the cost of energy imports. The minimization of this objective function, subject to the constraints describing the energy system gives an optimal investment plan for the selected technologies. The version of MARKAL used in the application was written in GAMS. It describes the Ontario energy supply system [24]. Figure 12 presents the Jacobian matrix as generated by GAMS. This display hides

a primal-block angular structure generated by the time scale dimension of the problem.

The subproblems correspond to activity and constraints within 5-year periods. The subproblems are linked by capacity transfer constraints. Giving this sole information to SPI allows one to retrieve the primal-block angular structure displayed on Figure 13. At the present, we have specified that the model should have 9 subproblems, one per period. Being suspicious that the MARKAL model enjoyed the structure of Figure 13, we passed two sets of instructions to SPI. The first instruction has the form of regular expression where the lines represent lengthy expressions that allow SPI to locate the position of ——————(*date*)—————— in the name retrieved from the dictionary. "date" runs through 9 5-year periods and retrieves data that belongs to one of 9 subproblems. The second instruction contains regular expressions to match the capacity transfer constraints: this determines data that belongs to the coupling constraints (master problem).

Naturally, this implementation was tailored to GAMS dictionary. Refining it or adapting it to other dictionaries is a pure implementation issue.

The complete model has 2603 rows and 2700 columns. The master problem has 348 rows and 2700 columns. The subproblems have in average 250 rows and 300 columns. Once the reordering is obtained, the selected SES (here the decomposition algorithm) computes optimal prices for the coupling constraints. There are several ways to retrieve a primal optimal solution. One way is to compute the amount of common resources allocated to each subproblem and solve each subproblem independently subject to this extra allocation constraint. The permutations P and Q resulting from the mappings $M_{\mathcal{R}}$ and $M_{\mathcal{C}}$ provided by SPI are used to send back to GAMS the solution in the original order. The final output is generated through GAMS, in the same way as it would be in the standard usage of this AML.

Let us conclude with a short comment on the final result. We had no insight into the particularities of the MARKAL model which was made available to us. We just suspected that the model should involve, like other MARKAL models, several periods. Using SPI, we have been able to elicit a primal block-angular structure. This structure fits a Lagrangian decomposition approach, and we used a special interior point decomposition solver to exploit it. However, the reader should be aware that, for the size of the example, a decomposition approach cannot compete with a frontal approach. Therefore, there was no point in reporting iteration counts or CPU measurements. Naturally, the SES reaches the correct optimal value and generates complete primal and dual optimal solutions. The points of interest in implementing SET on this example are as follows.

1. For the user, the only difference in using SET against a straight AML lies in SPI.

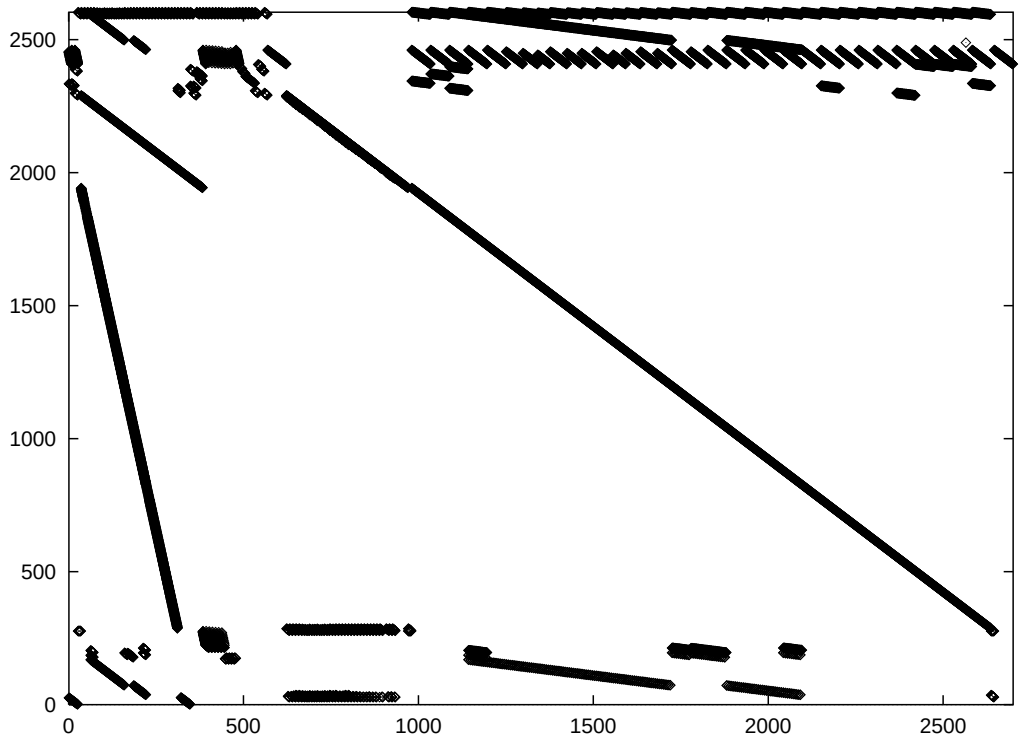


Figure 12: Anonymous matrix generated by GAMS for the MARKAL problem.

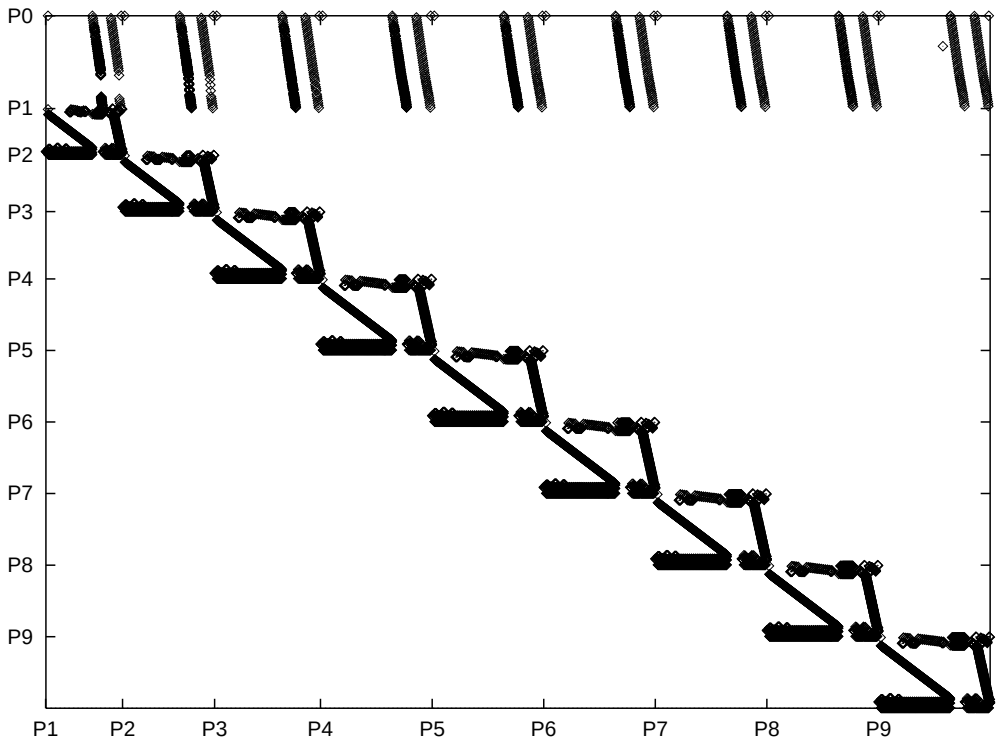


Figure 13: Reordered matrix for the MARKAL problem.

2. The amount of work needed to feed SPI with the proper information is much less than that required by the model formulation with the AML.

3. SET automatizes the links with the solver. If the file '`model.set`' is not present, then SET uses a standard LP solver hooked to GAMS; otherwise, SET uses our in-house interior point based decomposition method.

4. SET is totally transparent for the user. In particular, the user retrieves and manipulates the information in the optimal solution with the AML in the same way as if SET were not present.

6 Conclusion

In this paper we presented a new concept of the structure exploiting tool that enables model builders who use AMLs to generate complex models, to retrieve any structure embedded within a model and to pass it to a specialized solver. In this way, the user can hook the AML of choice to an appropriate structure exploiting solver. The specification of the structure and the use of SPI is significantly simpler than building the model itself. We believe that our concept and methodology will give to large audience access to the large body of advanced algorithmic tools.

References

[1] E. Andersen, J. Gondzio, C. Meszaros, and X. Xu, *Implementation of interior point methods for large scale linear programming*, in Interior Point Methods in Mathematical Programming, T. Terlaky, ed., Kluwer Academic Publishers, 1996, pp. 189–252.

[2] P. Barton, E. M. B. Smith, and C. C. Pantelides, *Combined discrete/continous process modelling using gPROMS*. Centre for Process Systems Engineering, Imperial College of Science, Technology and Medicine, London SW7 2AZ, United Kingdom, Prepared for presentation at the 1991 AIChE Annual Meeting, 1991.

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

[4] J. R. Birge, *Decomposition and partitioning methods for multistage stochastic linear programs*, Operations Research, 33 (1985), pp. 989–1007.

[5] J. R. Birge and L. Qi, *Computing block-angular Karmarkar projections with applications to stochastic programming*, Management Science, 34 (1988), pp. 1472–1479.

[6] A. Brooke, D. Kendrick, and A. Meeraus, *GAMS: A User's Guide*, The Scientific Press, Redwood City, California, 1992.

[7] D. Chang and E. Fragnière, *Splitdat and decomp: Two new GAMS I/O subroutines to handle mathematical programming problems with an automated decomposition procedure.* Stanford University, Department of Operations Research, manuscript, August 1996.

[8] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, 1963.

[9] G. B. Dantzig and P. Wolfe, *The decomposition algorithm for linear programming*, Econometrica, 29 (1961), pp. 767–778.

[10] R. Entriken, *The Parallel Decomposition of Linear Programs*, PhD thesis, Stanford University, 1989.

[11] R. Entriken, G. Infanger, and M. Saunders, *OMI: An application programming interface for optimization modeling.* Stanford University, Department of Operations Research, manuscript, October 1996.

[12] M. Ferris and J. Horn, *Partitioning mathematical programs for parallel solution*, tech. rep., Computer Sciences Department, University of Wisconsin–Madison, May 1994.

[13] R. Fourer, *Staircase matrices and systems*, SIAM Review, 26 (1983), pp. 1–70.

[14] R. Fourer, D. Gay, and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, The Scientific Press, San Francisco, California, 1993.

[15] E. Fragnière, J. Gondzio, and R. Sarkissian, *Efficient management of multiple sets to extract complex structures from mathematical programs*, tech. rep., Logilab, Section of Management Studies, University of Geneva, 102 Bd Carl Vogt, CH-1211 Geneva 4, Switzerland, March 1998.

[16] GAMS Development Corporation, *The GAMS I/O Library*, 1996.

[17] A. M. Geoffrion, *The SML language for structured modeling*, Operations Research, 40 (1992), pp. 38–75.

[18] J. Gondzio, *HOPDM (version 2.12) – a fast LP solver based on a primal-dual interior point method*, European Journal of Operational Research, 85 (1995), pp. 221–225.

[19] J. Gondzio, O. du Merle, R. Sarkissian, and J.-P. Vial, *ACCPM - a library for convex optimization based on an analytic center cutting plane method*, European Journal of Operational Research, 94 (1996), pp. 206–211.

[20] J. Gondzio and J.-P. Vial, *Warm start and ε-subgradients in cutting plane scheme for block-angular linear programs*, tech. rep., Logilab, University of Geneva, 102 Bd Carl-Vogt, CH-1211, June 1997, revised in November 1997. To appear in *Computational Optimization and Applications*.

[21] K. A. Hargreaves and K. Berry, *Regex*, Free Sotfware Foundation, 0.12a ed., September 1992.

[22] B. W. Kernighan, *UNIX for beginners - second edition*. AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1986.

[23] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partioning graphs*, The Bell System Technical Journal, (1970), pp. 291–307.

[24] R. Loulou and D. Lavigne, *MARKAL model with elastic demands: Application to greenhouse gas emission control*, in Operations Research and Environmental Management, C. Carraro and A. Haurie, eds., Dordrecht, The Netherlands, 1996, Kluwer Academic Publishers, pp. 201–220.

[25] E. Loute and J.-P. Vial, *A parallel block Cholesky factorization for staircase linear programs*, tech. rep., Center for Operations Research and Econometrics, Louvain, Belgium, 1993.

[26] A. S. Manne, R. Mendelson, and R. G. Richels, *MERGE - a model for evaluating regional and global effects of ghg reduction policies*, Energy Policy, 23 (1995), pp. 17–34.

[27] I. Maros and C. Mészáros, *The role of the augmented system in interior point methods*, European Journal of Operational Research, 107 (1998), pp. 720–736.

[28] P. Piela, T. G. Epperly, and K. M. Weterberg, *Ascend: an object-oriented computer environment for modeling and analysis: The modeling language*, Computers and Chemical Engineering, 15 (1991), pp. 53–72.

[29] F. P. Ramsey, *A mathematical theory of saving*, Economics Journal, (1928).

[30] C. Roos, T. Terlaky, and J.-P. Vial, *Interior Point Methods for Mathematical Programming*, John Wiley and Sons, New York, 1997.

[31] G. Stephanopoulos, G. Henning, and H. Leone, *Model.la: A modeling language for process engineering. Part I: the formal framework. Part II: Multifaceted modeling of process systems*, Computers and Chemical Engineering, 14 (1990), pp. 813–869.