AIMMS Language Reference - Implementing Advanced Algorithms for Mathematical Programs

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit www.aimms.com.

Copyright © 1993–2018 by AIMMS B.V. All rights reserved.

AIMMS B.V.
Diakenhuisweg 29-35
2033 AP Haarlem
The Netherlands
Tel.: +31 23 5511512

AIMMS Inc.
11711 SE 8th Street
Suite 303
Bellevue, WA 98005
USA
Tel.: +1 425 458 4024

AIMMS Pte. Ltd.
55 Market Street #10-00
Singapore 048941
Tel.: +65 6521 2827

AIMMS
SOHO Fuxing Plaza No.388
Building D-71, Level 3
Madang Road, Huangpu District
Shanghai 200025
China
Tel.: ++86 21 5309 8733

Email: info@aimms.com
WWW: www.aimms.com

Aimms is a registered trademark of AIMMS B.V. IBM ILOG CPLEX and CPLEX is a registered trademark of IBM Corporation. GUROBI is a registered trademark of Gurobi Optimization, Inc. Knitro is a registered trademark of Artelys. Windows and Excel are registered trademarks of Microsoft Corporation. TEX, LATEX, and $\mathcal{AMS}$-LATEX are trademarks of the American Mathematical Society. Lucida is a registered trademark of Bigelow & Holmes Inc. Acrobat is a registered trademark of Adobe Systems Inc. Other brands and their products are trademarks of their respective holders.

Information in this document is subject to change without notice and does not represent a commitment on the part of AIMMS B.V. The software described in this document is furnished under a license agreement and may only be used and copied in accordance with the terms of the agreement. The documentation may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from AIMMS B.V.

AIMMS B.V. makes no representation or warranty with respect to the adequacy of this documentation or the programs which it describes for any particular purpose or with respect to its adequacy to produce any particular result. In no event shall AIMMS B.V., its employees, its contractors or the authors of this documentation be liable for special, direct, indirect or consequential damages, losses, costs, charges, claims, demands, or claims for lost profits, fees or expenses of any nature or kind.

In addition to the foregoing, users should recognize that all complex software systems and their documentation contain errors and omissions. The authors, AIMMS B.V. and its employees, and its contractors shall not be responsible under any circumstances for providing information or corrections to errors and omissions discovered at any time in this book or the software it describes, whether or not they are aware of the errors or omissions. The authors, AIMMS B.V. and its employees, and its contractors do not recommend the use of the software described in this book for applications in which errors or omissions could threaten life, injury or significant loss.

This documentation was typeset by AIMMS B.V. using LATEX and the Lucida font family.

Chapter 16

Implementing Advanced Algorithms for Mathematical Programs

Control over the solution process

The SOLVE statement discussed in Section 15.3 offers a convenient way to execute all necessary steps to generate and solve a single instance of a mathematical program in one simple statement. For most applications, this level of control over the individual steps required to execute the generation and solution process is sufficient. However, for advanced applications, you may need a finer-grained level of control, e.g. to

- work with multiple, differing, instances of a single symbolic mathematical program,
- manipulate the individual rows and columns and the coefficient matrix of a mathematical program instance, for example to efficiently implement a column generation scheme,
- work with a repository of solutions associated with a mathematical program instance, for instance as a means to store multiple starting solutions or, within a solver callback, to setup and update a collection of incumbents of a mixed integer model, or
- start multiple solver sessions for a mathematical program instance, either locally or remotely.

This chapter

This chapter describes a library of procedures that offers you fine-grained control over the generation, manipulation and solution of a mathematical program instance, and allows you to manage a collection of solutions and solver sessions associated with such mathematical program instances. As you will see later on, the SOLVE statement can be completely expressed in terms of the procedures in this library.

16.1 Introduction to the GMP library

Introduction

With every MathematicalProgram declared as part of your model, the GMP library allows you to associate

- one or more *Generated Math Program instances* (GMPs),

and with each GMP

- a conceptual matrix of coefficients that can be manipulated,
- a repository of initial, intermediate or final solutions, and
- a pool of local or remote solver sessions.

Figure 16.1 illustrates the interrelationship between symbolic mathematical programs and the concepts of the GMP library, as well as the main properties that can be associated with each of them.

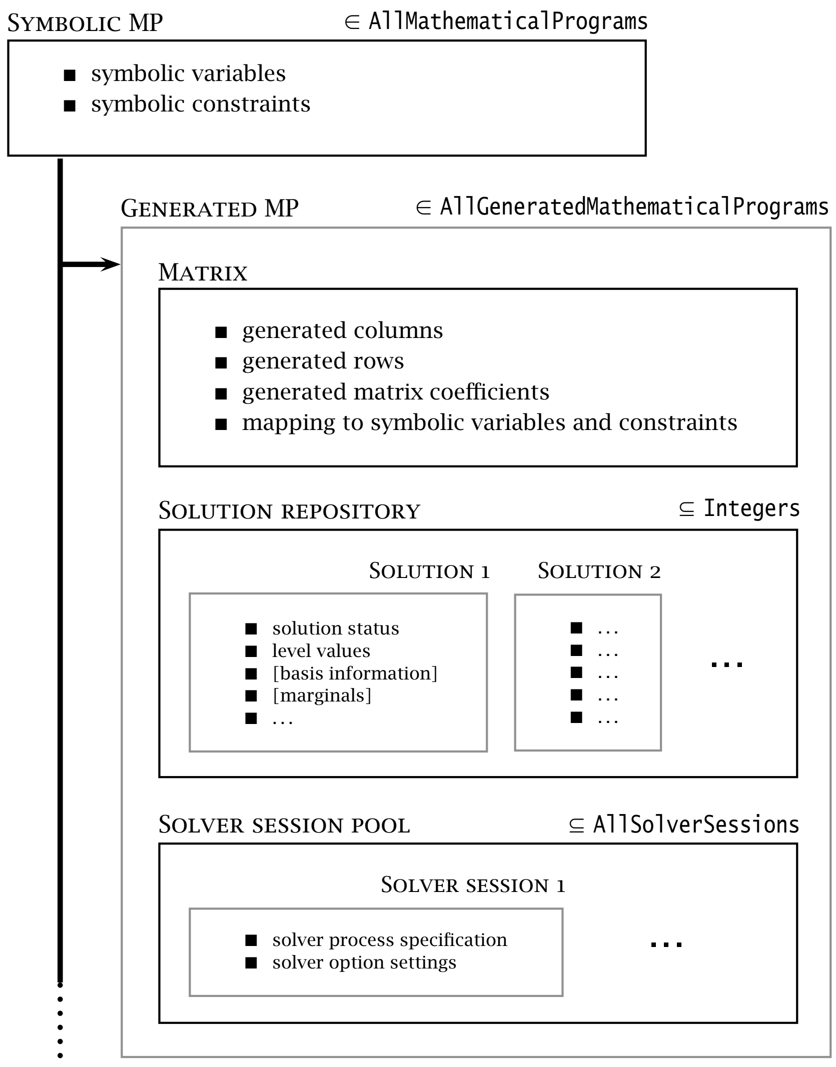

Figure 16.1: Concepts associated with a GMP

Generated mathematical program instances

For every `MathematicalProgram` declaration in your model, modifications in the index sets and input data referenced in constraints and variable definitions may give rise to completely different instances of the coefficient matrix when the mathematical program at hand is being generated.

An example: indexed instances

An illustrative example of such differing instances occurs when the constraints and variables of a symbolic mathematical program are indexed over a subset of some other superset. If you let the subset contain a single element of the superset, the generated instances will be completely different for each element of the superset. The effect of changing the contents of the subset in this manner, would almost compare to having an indexed `MathematicalProgram` *declaration* (which AIMMS does not support). In the worked example of Section 16.12.1 you will see, however, how you can obtain an indexed collection of generated mathematical program *instances* using the GMP library.

Need for multiple instances

With the standard `SOLVE` statement (see Section 15.3) you only have access to a single generated mathematical program instance for every symbolic mathematical program, namely the instance associated with the last call to the `SOLVE` statement for that particular mathematical program. This effectively eliminates the capability to efficiently implement an algorithm which requires the interaction between two or more generated instances of the same symbolic mathematical program. For this reason, the GMP library allows you to maintain and work with a collection of generated mathematical program instances simultaneously.

Matrix manipulation

The GMP library also allows you to manipulate the rows, columns and coefficients of the matrix of a mathematical program instance once it has been generated. If the number of modifications is relatively small, manipulating the matrix directly will save a considerable amount of time compared to letting AIMMS completely regenerate the matrix again through the standard `SOLVE` statement. You can use matrix manipulation, for instance

- to quickly add columns, and adapt the existing rows of the matrix accordingly, in a column generation scheme, or
- to dynamically add cuts to a mixed integer linear program.

Keeping multiple solutions

With the standard `SOLVE` statement, you only have access to a single solution of a mathematical program, namely the one stored in the symbolic variables and constraints that make up the mathematical program. There are, however, many situations where it would be convenient to have access to a repository of solutions. A solution repository can be used, for instance

- to store a collection of starting solutions for a NLP or MINLP problem. Solving the problem, in either a serial or parallel manner, with each of these starting solutions may help you find a better solution than by simply solving the problem with only a single starting solution.

- during the solution process of a mixed integer program, if you are interested in other integer solutions than the final solution returned by the solver. You can use the solution repository to store a fixed size collection of the best incumbent solutions returned by the solver during the solution process.

Solution repository

The GMP library comes with a solution repository for each generated mathematical program instance, and offers a number of functions to easily transfer a solution from and to either

- the data of the variables and constraints that make up the associated mathematical program in your model, or
- any solver session (explained below) associated with the generated mathematical program instance.

In fact, in the GMP library there is no direct solution/starting point transfer between a solver and the model, but such transfer always takes place through the solution repository.

Solver session pool

The final concept that is part of the GMP library is that of solver sessions. In principle, the GMP library is prepared to allow a generated mathematical program instance to keep a pool of associated solver sessions, each possibly set up with a different solver, or with different solver settings, and to be run either locally or remotely.

When useful

Using multiple solver session it becomes possible, for example, to let the same (or another) solver with different solver settings solve a mixed integer program instance in parallel, and pass tighter bound information found by one solver session to the other sessions by means of a callback implemented in your model.

GMP namespace

To prevent naming conflicts, all functions and procedure in the GMP library are member of the predefined `GMP` namespace. The `GMP` namespace is further partitioned into the namespaces

- `GMP::Instance`,
- `GMP::Row`,
- `GMP::Column`,
- `GMP::Coefficient`,
- `GMP::Event`,
- `GMP::QuadraticCoefficient`,
- `GMP::Solution`,
- `GMP::SolverSession`,
- `GMP::Stochastic`,
- `GMP::Robust`,
- `GMP::Benders`,

- `GMP::Linearization`, and
- `GMP::ProgressWindow`.

In the following sections we will discuss the procedures and functions contained in each of these namespaces.

Return values

When using the GMP library, it may be particularly important to check for any kind of error conditions that can occur. To help you catch such errors, the procedures and functions in the `GMP` namespace either return

- a 1 when successful, or 0 otherwise (for procedures), or
- a non-empty element in one of the GMP-related predefined sets when successful, or the empty element otherwise (for functions).

Note that, for the sake of brevity, most of the examples in this chapter do not perform error checking of any kind.

16.2 Managing generated mathematical program instances

Managing math program instances

The procedures and functions of the `GMP::Instance` namespace are listed in Table 16.1 and take care of the creation and management of generated mathematical program instances. Mathematical program instances also provide access to the solution repository and solver sessions associated with the instance.

Creation of mathematical program instances

New mathematical program instances can be created by calling

- the `SOLVE` statement,
- the `GMP::Instance::Generate` function,
- the `GMP::Instance::GenerateRobustCounterpart` function,
- the `GMP::Instance::GenerateStochasticProgram` function,
- the `GMP::Instance::Copy` function,
- the `GMP::Instance::CreateDual` function,
- the `GMP::Instance::CreateFeasibility` function,
- the `GMP::Instance::CreatePresolved` function,
- the `GMP::Instance::CreateMasterMIP` function,
- the `GMP::Stochastic::CreateBendersRootproblem` function,
- the `GMP::Stochastic::BendersFindFeasibilityReference` function, or
- the `GMP::Stochastic::BendersFindReference` function.

Generate(*MP*, *name*)→AllGeneratedMathematicalPrograms
Copy(*GMP*, *name*)→AllGeneratedMathematicalPrograms
Rename(*GMP*, *name*)
Delete(*GMP*)	

GenerateRobustCounterpart(*MP*, *UncertainParameters*, *UncertaintyConstraints* [, *Name*])→AllGeneratedMathematicalPrograms	
GenerateStochasticProgram(*MP*, *StochasticParameters*, *StochasticVariables*, *Scenarios*, *ScenarioProbability*, *ScenarioTreeMap*, *RootScenario* [, *GenerationMode*][, *Name*])→AllGeneratedMathematicalPrograms	

CreateMasterMIP(*GMP*, *name*)→AllGeneratedMathematicalPrograms
FixColumns(*GMP1*, *GMP2*, *solNr*, *varSet*)
AddIntegerEliminationRows(*GMP*, *solNr*, *elimNo*)
DeleteIntegerEliminationRows(*GMP*, *elimNo*)	

CreateDual(*GMP*, *name*)→AllGeneratedMathematicalPrograms CreateFeasibility(*GMP*[, *name*][, *useMinMax*])→AllGeneratedMathematicalPrograms CreatePresolved(*GMP*, *name*)→AllGeneratedMathematicalPrograms	

GetSymbolicMathematicalProgram(*GMP*)→AllMathematicalPrograms
GetNumberOfRows(*GMP*)
GetNumberOfColumns(*GMP*)
GetNumberOfNonzeros(*GMP*)	

GetDirection(*GMP*)→AllMathematicalProgrammingDirections SetDirection(*GMP*, *dir*)	

GetOptionValue(*GMP*, *OptionName*)
SetOptionValue(*GMP*, *OptionName*, *Value*)	

CreateProgressCategory(*GMP*[, *Name*])→AllProgressCategories	
GetMathematicalProgrammingType(*GMP*)→AllMathematicalProgrammingTypes SetMathematicalProgrammingType(*GMP*, *type*)	
GetSolver(*GMP*)→AllSolvers SetSolver(*GMP*, *solver*)	

SetCallbackAddCut(*GMP*, *CB*)
SetCallbackBranch(*GMP*, *CB*)
SetCallbackNewIncumbent(*GMP*, *CB*)
SetCallbackHeuristic(*GMP*, *CB*)
SetCallbackTime(*GMP*, *CB*)
	SetCallbackAddLazyConstraint(*GMP*, *CB*)
	SetCallbackIncumbent(*GMP*, *CB*)
	SetCallbackStatusChange(*GMP*, *CB*)
	SetCallbackIterations(*GMP*, *CB*, *nrIters*)

SetIterationLimit(*GMP*, *nrIters*)
SetTimeLimit(*GMP*, *nrSeconds*)
	SetMemoryLimit(*GMP*, *nrMB*)
	SetCutoff(*GMP*, *value*)

Solve(*GMP*) FindApproximatelyFeasibleSolution(*GMP*, *sol1*, *sol2*, *nrIter*[, *maxIter*][, *feasTol*] [, *moveTol*][, *imprTol*][, *maxTime*][, *useSum*][, *augIter*][, *useBest*])	

GetObjective(*GMP*)
GetBestBound(*GMP*)
GetMemoryUsed(*GMP*)
MemoryStatistics(*GMPSet*, *OutputFileName*[, *optional-arguments* . . .])	

GetColumnNumbers(*GMP*, *varSet*)→Integers GetRowNumbers(*GMP*, *conSet*)→Integers GetObjectiveColumnNumber(*GMP*)→Integers GetObjectiveRowNumber(*GMP*)→Integers	
CreateSolverSession(*GMP*[, *Name*][, *Solver*])→AllSolverSessions DeleteSolverSession(*solverSession*)	

Table 16.1: Procedures and functions in GMP::Instance namespace

All mathematical program instances created through each of these calls, are uniquely represented by elements in the predefined set `AllGeneratedMathematicalPrograms`. For the functions in the `GMP::Instance` namespace creating GMPs you can explicitly specify the name of the associated set element to be created. When calling the `SOLVE` statement, AIMMS will generate an element with the same name as the `MathematicalProgram` at hand. When the name of the element to be created is already contained in the set `AllGeneratedMathematicalPrograms`, the mathematical program instance associated with the existing element will be completely replaced by the newly created mathematical program instance.

Special math programming types

Stochastic programming and the use of the function `GenerateStochasticProgram` is discussed in Section 19.4. Robust optimization and the use of the function `GenerateRobustCounterpart` is explained in Section 20.5. The functionality of the `CreateDual` function is explained in more detail in Section 16.2.1. The function `CreateMasterMIP` is used by the AIMMS Outer Approximation solver, which is discussed in full detail in Chapter 18. Presolving of mathematical programs is discussed in Section 17.1.

Deleting and renaming instances

Through the procedures `GMP::Instance::Delete` and `GMP::Instance::Rename` you can delete and rename mathematical program instances and their associated elements in the set `AllGeneratedMathematicalPrograms`. If you rename a mathematical program instance to a name that already exists in the set `AllGeneratedMathematicalPrograms`, the associated mathematical program instance will be deleted prior to renaming.

CLEANDEPENDENTS statement

Note that also the `CLEANDEPENDENTS` statement may remove mathematical program instances from memory when it affects any constraint or variable referenced by that instance.

Retrieving and setting basic properties

Through the functions

- `GMP::Instance::GetSymbolicMathematicalProgram`,
- `GMP::Instance::GetNumberOfRows`,
- `GMP::Instance::GetNumberOfColumns`,
- `GMP::Instance::GetNumberOfNonzeros`,
- `GMP::Instance::GetDirection`, and
- `GMP::Instance::GetMathematicalProgrammingType`

you can retrieve the current value of some basic properties of a mathematical program instance. The number of rows, columns and nonzeros can be changed by manipulating the matrix of the mathematical program instance (see also Section 16.3). You can use the functions

- `GMP::Instance::SetDirection`, and
- `GMP::Instance::SetMathematicalProgrammingType`

to modify the optimization direction and mathematical programming type. The type of a mathematical program must be a member of the set `MathematicalProgrammingTypes` (see also section 15.1) The direction associated with a mathematical program is either

- `'maximize'`,
- `'minimize'`, or
- `'none'`.

The direction `'none'` is the instruction to the solver to find a feasible solution.

Installing callbacks

For each mathematical program instance, you can set up to six callback functions that will be called by any solver session associated with the mathematical program instance at hand. Through the following procedures you can install or uninstall a callback function for a mathematical program instance.

- `GMP::Instance::SetCallbackAddCut`
- `GMP::Instance::SetCallbackAddLazyConstraint`
- `GMP::Instance::SetCallbackBranch`
- `GMP::Instance::SetCallbackIncumbent`
- `GMP::Instance::SetCallbackNewIncumbent`
- `GMP::Instance::SetCallbackStatusChange`
- `GMP::Instance::SetCallbackHeuristic`
- `GMP::Instance::SetCallbackIterations`
- `GMP::Instance::SetCallbackTime`

Each of these procedures expects an element of the set `AllProcedures`, or an empty element `''` to uninstall the callback.

Callback procedures

Callback procedures for each type of callback should be declared as follows:

`AnExampleCallback`(*solverSession*)

where the *solverSession* argument should be a scalar input element parameter into the set `AllSolverSessions`. Callback procedures should have a return value of

- 0, if you want the solver session to stop, or
- 1, if you want the solver session to continue.

As discussed before, each solver session can be uniquely associated with a single mathematical program instance. You can find this instance by calling the function `GMP::SolverSession::GetInstance` (see also Section 16.5), and, within the callback procedure, use this instance to get access to its associated properties.

Example

The following example implements a callback procedure for the new incumbent callback. The callback procedure finds the associated mathematical program instance, and stores all incumbents reported by the solver into the next solution of the solution repository.

```
Procedure IncumbentCallBack {
    Arguments  : solvSess;
    Body       : {
        theGMP := GMP::SolverSession::GetInstance( solvSess );
        GMP::Solution::RetrieveFromSolverSession( solvSess, solutionNumber(theGMP) );
        solutionNumber(theGMP) += 1;

        return 1;   ! continue solving
    }
}
```

Note that the callback procedure uses the `GMP::Solution::RetrieveFromSolverSession` function (discussed in Section 16.4) to retrieve the solution from the solver.

Solving mathematical program instances

In contrast to the `SOLVE` statement, the philosophy behind the GMP library is to break down the optimization functionality in AIMMS to a level which offers optimum support for implementing advanced algorithms around a `MathematicalProgram` in your model. One of the consequences of this philosophy is that the solution is never directly transferred between the symbolic variables and constraints and the solver, but is intermediately stored in a solution repository. Therefore, solving a `MathematicalProgram` using the GMP library breaks down into the following basic steps:

1. generate a mathematical program instance for the `MathematicalProgram`,
2. create a solver session for the mathematical program instance,
3. transfer the initial point from the model to the solution repository,
4. transfer the initial point from the solution repository to the solver session,
5. let the solver session solve the problem,
6. transfer the final solution from the solver session to the solution repository, and
7. transfer the final solution from the solution repository to the model.

Solving the instance directly

For your convenience, however, the GMP library contains a procedure

- `GMP::Instance::Solve`

which, given a generated mathematical program instance, takes care of all intermediate steps (i.e. steps 2-7) necessary to solve the mathematical program instance. In case you need access to the solution in the solution repository after calling the `GMP::Instance::Solve` call, you should notice that the `GMP::Instance::Solve` procedure (as well as the `SOLVE` statement) performs all of its solution transfer through the fixed solution number 1 in the solution repository.

Emulating the SOLVE statement

The following AIMMS code provides an emulation of the SOLVE statement in terms of GMP::Instance functions.

```
! Generate an instance of the mathematical program MPid and add
! the element 'MPid' to the set AllGeneratedMathematicalPrograms.
! This element is returned into the element parameter genGMP.
genGMP := GMP::Instance::Generate(MPid, FormatString("%e", MPid));

! Actually solve the problem using the solve procedure for an
! instance (which communicates through solution number 1).
GMP::Instance::Solve(genGMP);
```

Multistart support

The function FindApproximatelyFeasibleSolution is used by the AIMMS multistart algorithm (see Section 17.2) to compute an approximately feasible solution for an NLP problem. The algorithm used by this function to find the approximately feasible solution is described in [Ch04].

Creating solver sessions

For each generated mathematical program instance, you can explicitly create and delete one or more solver sessions using the following functions:

- GMP::Instance::CreateSolverSession, and
- GMP::Instance::DeleteSolverSession.

Once created, you can use the solver session to solve the generated mathematical program

- in a blocking manner by calling the GMP::SolverSession::Execute function, or
- in a non-blocking manner by calling the GMP::SolverSession::AsynchronousExecute function.

Prior to calling the GMP::SolverSession::Execute or GMP::SolverSession::AsynchronousExecute functions, you should call the function GMP::Solution::SendToSolverSession to initialize the solver session with a solution stored in the solution repository. Using an explicit solver session allows you, for instance, to solve an NLP problem with several initial solutions stored in the solution repository.

Multiple sessions allowed

AIMMS allows you to create multiple solver sessions per mathematical program instance, and solve them in parallel. You can solve multiple mathematical program instances in parallel, by calling the function GMP::SolverSession::AsynchronousExecute multiple times. The function starts a separate thread of execution to solve the math program instance asynchronously, and returns immediately. To solve multiple mathematical program instances in parallel, your computer should have multiple processors or a multi-core processor.

Deleting solver sessions

Once the function GMP::SolverSession::Execute or GMP::SolverSession::AsynchronousExecute has been called, the internal solver representation of the mathematical program instance will be created. The solver representation will only be deleted—and its associated resources freed—when the corresponding solver session has been deleted by calling the function GMP::Instance::DeleteSolverSession.

Implementing the procedure GMP::Instance::Solve

The GMP:Instance::Solve procedure discussed previously can be emulated using solver sessions, as illustrated in the equivalent code below.

```
! Create a solver session for genMP, which will create an element
! in the set AllSolverSessions, and assign the newly created element
! to the element parameter session.
session := GMP::Instance::CreateSolverSession(genMP);

! Copy the initial solution from the variables in AIMMS to
! solution number 1 of the generated mathematical program.
GMP::Solution::RetrieveFromModel(genMP,1);

! Send the solution stored in solution 1 to the solver session
GMP::Solution::SendToSolverSession(session, 1);

! Call the solver session to actually solve the problem.
GMP::SolverSession::Execute(session);

! Copy the solution from the solver session into solution 1.
GMP::Solution::RetrieveFromSolverSession(session, 1);

! Store this solution in the AIMMS variables and constraints.
GMP::Solution::SendToModel(genMP, 1);
```

Setting default solver session limits

You can use the following procedures to set various default limits that apply to all solver sessions created through GMP::Instance::CreateSolverSession.

- GMP::Instance::SetIterationLimit
- GMP::Instance::SetMemoryLimit
- GMP::Instance::SetTimeLimit
- GMP::Instance::SetCutoff

Setting GMP-specific options

For every *GMP* you can override the default project options using the function GMP::Instance::SetOptionValue. You can also set options for a specific solver session associated with a *GMP* through the function GMP::SolverSession::SetOptionValue. In turn, option values set for a specific solver session override the option values for the associated *GMP*.

Setting the default solver

Similarly, you can get and set the default solver that will be used by all solver sessions created through GMP::Instance::CreateSolverSession.

- GMP::Instance::GetSolver
- GMP::Instance::SetSolver

Outer approximation support

Through the functions

- `GMP::Instance::CreateMasterMIP`
- `GMP::Instance::FixColumns`
- `GMP::Instance::AddIntegerEliminationRows`
- `GMP::Instance::DeleteIntegerEliminationRows`

the GMP library offers support for solving mixed integer nonlinear (MINLP) problems using a white box outer approximation approach. The AIMMS Outer Approximation solver is discussed in full detail in Chapter 18.

16.2.1 Dealing with degeneracy and non-uniqueness

Background

When solving a mathematical program, some practical difficulties may arise when the optimal solution of the underlying model is either degenerate and/or not unique (i.e. there are multiple optimal solutions). These difficulties may concern both the primal and dual solution (i.e. the shadow prices).

Problems with degeneracy

In the case of degeneracy (see also Section 4.2 of the AIMMS Modeling Guide for an explanation), the solution status of one or more variables is “basic at bound”. In the presence of degeneracy, shadow prices are no longer unique, and their interpretation is therefore ambiguous. As a result, if the shadow prices have an economic interpretation in the application, the particular shadow prices found by the solver cannot be presented to the end-user in a meaningful and reliable fashion.

Problems with multiple solutions

In the case of multiple solutions, the situation is even worse. There are multiple optimal bases, and the associated shadow prices differ between these bases (just as with degeneracy). In addition, the solution presented to the end-user is no longer unique, which may raise questions by the end-user as to why a particular solution is presented.

Degeneracy and multiple solutions

Both degeneracy and multiple solutions can occur at the same time, having their combined effect on the non-uniqueness of both the primal and the dual solution (the optimal shadow prices). The following two paragraphs present possible solutions to deal with multiple primal and dual solutions.

Towards a unique primal solution

One way to deal with multiple solutions is to find a new and second objective function specifically designed to deal with eliminating the multiplicity of solutions. This might be accomplished, for instance, by adding new sets of variables and constraints to cap some aspect of the primal model, and the maximum cap could then be minimized. Or perhaps a straightforward modification of the original objective function could become the second auxiliary objective. It is important to note that this second objective function is optimized only after the first objective function is fixed at its previous optimal value and has been added as a constraint.

Implementing primal uniqueness

Using the functionality provided by the GMP library, constructing a second objective function for a mathematical program is a straightforward task:

- generate and solve the original mathematical program,
- use the matrix manipulations procedures discussed in Section 16.3 to create a new objective and fix the original one in the associated mathematical program instance,
- resolve the modified mathematical program instance.

Towards a unique dual solution

In the presence of primal degeneracy and/or multiple primal solutions, it is impossible to influence the selection of shadow prices, as this decision is made by the solver. To give the control back to you as a model developer, the only sensible step is to go directly to the dual formulation, and work with the model expressed in terms of shadow prices. It is then possible to construct a second auxiliary objective function designed to produce economically meaningful shadow prices. Again, it is important to note that this second objective function is optimized only after the original objective function is fixed at the optimal objective function value of the primal model, and has been added as a constraint.

Creating a dual mathematical program instance

To support the procedure for reaching dual uniqueness, the GMP library contains the function

- `GMP::Instance::CreateDual`

which creates the dual mathematical program instance associated with a given primal mathematical program instance.

Standard dual formulation

For a mathematical program of the form

Minimize:

$$\sum_i c_i x_i$$

Subject to:

$$\begin{aligned} \sum_i A_{ij} x_i &\geq b_j \qquad \forall j \\ x_i &\geq 0 \qquad \forall i \end{aligned}$$

the dual mathematical program can be formulated as follows

Maximize:

$$\sum_j b_j \lambda_j$$

Subject to:

$$\sum_j A_{ij}\lambda_j \le c_i \qquad \forall i$$
$$\lambda_j \ge 0 \qquad \forall j$$

where the λ_j represent the shadow prices of the constraints of the primal formulation.

Sign changes

If the primal formulation contains nonpositive or free variables, or contains $\le$ or equality constraints, a number of simple substitution will bring the formulation back into the standard form above, after which the above dual formulation can be used directly. The resulting changes to the dual formulation are as follows:

- a nonpositive variable x_i corresponds to a dual $\ge$ constraint,
- a free variable x_i corresponds to a dual equality constraint,
- a $\le$ constraint corresponds to a nonpositive dual variable λ_j, and
- an equality constraint corresponds to a free dual variable λ_j.

Bounded variables and ranged constraints

However, such simple transformation are not possible anymore if the primal model contains:

- bounded variables, i.e. $l_i \le x_i \le u_i$, or
- ranged constraints, i.e. $d_i \le \sum_i A_{ij}x_i \le b_j$.

In these cases, additional constraints (implicitly) have to be added as follows to satisfy the above standard formulation:

- $x_i \ge l_i$ whenever $l_i \ne 0, -\infty$,
- $x_i \le u_i$ whenever $u_i \ne 0, \infty$, and
- $\sum_i A_{ij}x_i \ge d_j$.

In the generated dual mathematical program, such implicit constraint additions in the primal formulation will lead to the explicit introduction of additional variables in the dual formulation. Such variable additions to the dual formulation are taken care of by AIMMS automatically, but will have consequences when you want to manipulate the matrix of the dual mathematical program instance, as discussed in Section 16.3.6.

Implementing dual uniqueness

Using the function `GMP::Instance::CreateDual`, it is relatively straightforward to implement the procedure outlined above to reach dual uniqueness:

- generate and solve the original mathematical program,
- generate a dual mathematical program instance from the primal mathematical program instance,
- use the matrix manipulations procedures discussed in Section 16.3 to create a new dual objective and fix the original dual objective in the newly created dual mathematical program instance,
- solve the modified dual mathematical program instance.

16.3 Matrix manipulation procedures

Matrix manipulation

The matrix manipulation procedures in the GMP library allow you to implement efficient algorithms for generated mathematical program instances which require only slight modifications of the matrix associated with the mathematical program instance during successive runs. These procedures operate directly on the coefficient matrix underlying the mathematical program, and thus avoid the constraint-generation process normally initiated by the `SOLVE` statement after input data has been modified.

This section

Prior to discussing the individual matrix manipulation procedures, the following section will provide some motivation when and when not to use matrix manipulation.

16.3.1 When to use matrix manipulation

When to use matrix manipulation

Even though AIMMS offers a library of matrix manipulation procedures, you should not use them blindly. As explained below, it is important to distinguish between manual and automatic input data changes inside an AIMMS application. Your decision whether or not to use the matrix manipulation procedures described in this section, should depend on this distinction.

Manual data input . . .

Consider an end-user of an AIMMS application who, after having looked at the results of a mathematical program, wants to make changes in the input data and then look again at the new solution of the mathematical program. The effect of the data changes on the input to the solver cannot be predicted in advance. Even a single data change could lead to multiple changes in the input to the solver, and could also cause a change in the number of constraints and variables inside the particular mathematical program.

. . . requires structure recognition

As a result, AIMMS has to determine whether or not the structure of the underlying mathematical program has changed. Only then can AIMMS decide whether the value of existing coefficients can be overwritten, or whether a new and structurally different data set has to be provided to the solver. This structure recognition step is time consuming, and cannot be avoided in the absence of any further information concerning the changes in input data.

Automatic data input . . .

Whenever input data are changed inside an AIMMS procedure, their effect on the input to the solver can usually be determined in advance. This effect may be nontrivial, in which case it is not worth the effort to establish the consequences. Rather, letting AIMMS perform the required structure recognition step through the regular `SOLVE` statement before passing new information to the solver seems to be a better remedy. There are several instances, however, in which the effect of data changes on the solver input data is easy to determine.

. . . may reflect particular structure

Consider, for instance, automatic data changes that have a one-to-one correspondence with values in the underlying mathematical program. In these instances, the incidence of variables in constraints is not modified, and only the replacement values of some coefficients need to be supplied to the particular solver. Other examples include automatic data changes that could create new values for particular variable-constraint combinations, or that could even cause new constraints or variables to be added to the input of the solver. In all these instances, the exact effects on the input of the solver can easily be determined in advance, and there is no need to let AIMMS perform of the computationally intensive structure recognition step of the `SOLVE` statement before passing new information to the solver.

Restrictions on usage

The above effects of data input modifications on the input to the solver are straightforward to implement with linear and quadratic mathematical programs, because the underlying data structures are matrices with rows, columns and nonzero elements. The input data structures for nonlinear mathematical programs are essentially nonlinear expressions. Modifications of the type discussed in the previous paragraph are not easily passed onto these nonlinear data structures. For this reason, the efficient updating of solver input has been confined to

- linear and quadratic constraints, and
- coefficients of nonlinear constraints with respect to variables that only occur linearly in that constraint.

Regeneration of nonlinear constraints

Whenever the input data of a nonlinear expression in a nonlinear constraint has changed, it is not possible anymore to change the nonlinear expression used by the solver directly to reflect the data change. You can still request AIMMS to regenerate the entire row, which will then use the updated inputs. You should note, however, that any modifications to the linear part of the regenerated constraint are lost after the constraint has been regenerated.

Scalar arguments only

All matrix procedures listed in Tables 16.2–16.5 have scalar-valued arguments. The *row* argument should always be

- a scalar reference to an existing constraint name in your model, or
- a row number which is an integer in the range $\{0..m-1\}$ whereby m is the number of rows.

The *column* argument should always be

- a scalar reference to an existing variable name in your model, or

- a column number which is an integer in the range $\{0..n-1\}$ whereby n is the number of columns.

Mathematical program instance required

Before you can apply any of the procedures of Tables 16.2-16.5, you must first create a mathematical program instance using any of the functions for this purpose discussed in Section 16.2. Either of these methods will set up the initial row-column matrix required by the matrix manipulation procedures. Also, any row or column referenced in the matrix manipulation procedures must either have been generated during the initial generation step, or must have been generated later on by a call to the procedures `GMP::Row::Add`, or `GMP::Column::Add`, respectively.

16.3.2 Coefficient modification procedures

Coefficient modification procedures

The procedures and functions of the `GMP::Coefficient` namespace are listed in Table 16.2 and take care of the modification of coefficients in the matrix and objective of a generated mathematical program instance.

`Get`(*GMP*, *row*, *column*)
`Set`(*GMP*, *row*, *column*, *value*)
`SetMulti`(*GMP*, *binding*, *row*, *column*, *value*)
`GetQuadratic`(*GMP*, *column1*, *column2*)
`SetQuadratic`(*GMP*, *column1*, *column2*, *value*)

Table 16.2: Procedures and functions in `GMP::Coefficient` namespace

Modifying coefficients

You can instruct AIMMS to modify any particular coefficient in a matrix by specifying the corresponding row and column (in AIMMS notation), together with the new value of that coefficient, as arguments of the procedure `GMP::Coefficient::Set`. This procedure can also be used when a value for the coefficient does not exist prior to calling the procedure. If many coeffients for a row or column have to be changed then it is more efficient to use `GMP::Coefficient::SetMulti`.

Quadratic coefficients

For quadratic mathematical programs, you can modify the quadratic objective coefficients by applying the function `GMP::Coefficient::SetQuadratic` to the objective row. For every two columns x_1 and x_2 you can specify the modified coefficient c_{12} if $c_{12}x_1x_2$ is to be part of the quadratic objective.

16.3.3 Quadratic coefficient modification procedures

Quadratic coefficient modification procedures

The procedures and functions of the GMP::QuadraticCoefficient namespace are listed in Table 16.3 and take care of the modification of coefficients of quadratic rows in the matrix other than the objective of a generated mathematical program instance.

Get(*GMP*, *row*, *column1*, *column2*)
Set(*GMP*, *row*, *column1*, *column2*, *value*)

Table 16.3: Procedures and functions in GMP::QuadraticCoefficient namespace

Modifying coefficients

You can instruct Aimms to modify any particular quadratic coefficient in a matrix by specifying the corresponding row and columns (in Aimms notation), together with the new value of that coefficient, as arguments of the procedure GMP::QuadraticCoefficient::Set. This procedure can also be used when a value for the quadratic coefficient does not exist prior to calling the procedure.

16.3.4 Row modification procedures

Row modification procedures

The procedures and functions of the GMP::Row namespace are listed in Table 16.4 and take care of the modification of properties of existing rows and the creation of new rows.

Row types

The row type refers to one of the four possibilities

- '<=',
- '=',
- '>=', and
- 'ranged'

You are free to change this type for each row. Deactivating and subsequently reactivating a row are instructions to the solver to ignore the row as part of the underlying mathematical program and then reconsider the row again as an active row.

Row generation

When you add a new row to a matrix using GMP::Row::Add, the newly added row will initially only have any zero coefficients, regardless of whether the corresponding Aimms constraint had a definition or not. Through the procedure GMP::Row::Generate you can tell Aimms to discard the current contents of a row in the matrix, and insert the coefficients as they follow from the definition of the corresponding constraint in your model.

`Add`(*GMP*, *row*)
`Delete`(*GMP*, *row*)
`Activate`(*GMP*, *row*)
`Deactivate`(*GMP*, *row*)
`Generate`(*GMP*, *row*)
`GetLeftHandSide`(*GMP*, *row*)
`SetLeftHandSide`(*GMP*, *row*, *value*)
`GetRightHandSide`(*GMP*, *row*)
`SetRightHandSide`(*GMP*, *row*, *value*)
`SetRightHandSideMulti`(*GMP*, *binding*, *row*, *value*)
`GetType`(*GMP*, *row*) → `AllRowTypes`
`SetType`(*GMP*, *row*, *type*)
`GetStatus`(*GMP*, *row*) → `AllRowColumnStatuses`
`DeleteIndicatorCondition`(*GMP*, *row*)
`GetIndicatorColumn`(*GMP*, *row*)
`GetIndicatorCondition`(*GMP*, *row*)
`SetIndicatorCondition`(*GMP*, *row*, *column*, *value*)
`GetConvex`(*GMP*, *row*)
`GetRelaxationOnly`(*GMP*, *row*)
`SetConvex`(*GMP*, *row*, *value*)
`SetRelaxationOnly`(*GMP*, *row*, *value*)
`SetPoolType`(*GMP*, *row*, *value*[, *mode*])
`SetPoolTypeMulti`(*GMP*, *binding*, *row*, *value*, *mode*)

Table 16.4: Procedures and functions in `GMP::Row` namespace

Indicator conditions

When you are using the CPLEX, GUROBI or ODH-CPLEX solver, you can declaratively specify indicator constraints through the `IndicatorConstraint` property of a constraint declaration (see Section 14.2.4). You can also set and delete indicator constraints programmatically for a given *GMP* using the functions `GMP::Row::SetIndicatorCondition` and `GMP::Row::DeleteIndicatorCondition`

Lazy and cut pool constraints

When you are using the CPLEX, GUROBI or ODH-CPLEX solver, you can declaratively specify constraints to be part of a pool of lazy constraints or cuts through the `IncludeInLazyConstraintPool` and `IncludeInCutPool` properties of a constraint declaration respectively (see Section 14.2.4). You can also specify lazy and cut pool constraints programmatically for a given *GMP* using the function `GMP::Row::SetPoolType`. (If the pool type has to be set for many rows then it is more efficient to use `GMP::Row::SetPoolTypeMulti`.)

Convex and relaxation-only constraints

Through the `.Convex` and `.RelaxationOnly` suffices of constraints you can set special constraint properties for the BARON global optimization solver (see also Section 14.2.6). For a given *GMP* you can also set these constraint properties programmatically using the `GMP::Row::SetConvex` and `GMP::Row::SetRelaxationOnly` functions.

16.3.5 Column modification procedures

The procedures and functions of the `GMP::Column` namespace are listed in Table 16.5 and take care of the modification of properties of existing columns and the creation of new columns.

`Add`(*GMP*, *column*)
`Delete`(*GMP*, *column*)
`Freeze`(*GMP*, *column*, *value*)
`FreezeMulti`(*GMP*, *binding*, *column*, *value*)
`Unfreeze`(*GMP*, *column*)
`UnfreezeMulti`(*GMP*, *binding*, *column*)
`GetLowerBound`(*GMP*, *column*)
`SetLowerBound`(*GMP*, *column*, *value*)
`SetLowerBoundMulti`(*GMP*, *binding*, *column*, *value*)
`GetUpperBound`(*GMP*, *column*)
`SetUpperBound`(*GMP*, *column*, *value*)
`SetUpperBoundMulti`(*GMP*, *binding*, *column*, *value*)
`GetType`(*GMP*, *column*) → `AllColumnTypes`
`SetType`(*GMP*, *column*, *type*)
`GetStatus`(*GMP*, *column*) → `AllRowColumnStatuses`
`SetDecomposition`(*GMP*, *column*, *value*)
`SetDecompositionMulti`(*GMP*, *binding*, *column*, *value*)
`SetAsObjective`(*GMP*, *column*)

Table 16.5: Procedures and functions in `GMP::Column` namespace

Column types

The column type refers to one of the three possibilities

- `'integer'`,
- `'continuous'`, and
- `'semi-continuous'`.

You are free to specify a different type for each column. For newly added columns, AIMMS will (initially) use the lower bound, upper bound and column type as specified in the declaration of the (symbolic) variable associated with the added column. Freezing a column and subsequently unfreezing it are instructions to the solver to fix the corresponding variable to its current value, and then free it again by letting it vary between its bounds.

Multi variants

If you want to change the data of many columns belonging to some variable then it is more efficient to use the multi variant of a procedure. A multi variant is available for freezing and unfreezing columns, and for setting the lower and upper bounds.

Changing the objective column

If you want to implement the procedures for reaching primal or dual uniqueness as described in Section 16.2.1, you can use the procedure

- `GMP::Column::SetAsObjective`

to change the objective function used by either the primal or dual mathematical program instance that you want to solve for a second time. Notice that the defining constraint for this variable should be

- part of the original mathematical program formulation for which AIMMS has generated a mathematical program instance, or
- added later on to the primal or dual generated mathematical program instance using the `GMP::Row::Add` procedure, where the row definition is generated by AIMMS through the `GMP::Row::Generate` procedure or constructed explicitly through several calls to the `GMP::Coefficient::Set` procedure.

16.3.6 Modifying an extended math program instance

Extended math program instances

To use the matrix manipulation routines of the GMP library, you must be able to associate every row and column of the matrix of the math program instance you want to manipulate with a symbolic constraint or variable within your model. However, some routines in the GMP library generate rows and columns that cannot be directly associated with specific symbolic constraints and variables in your model. Examples of such routines are:

- the `GMP::Instance::CreateDual` procedure, which may generate additional variables in the dual formulation for bounded variables and ranged constraints in the primal formulation (see also Section 16.2.1),
- the `GMP::Linearization::Add` and `GMP::Linearization::AddSingle` procedures, which add linearizations of nonlinear constraints to a specific math program instance (see also Section 16.10), and
- the `GMP::Instance::AddIntegerEliminationRows` procedure.

The rows and columns generated by these procedures can, however, be *indirectly* associated with symbolic constraints, variables or mathematical programs, as will be explained below.

Extended suffices

To support the use of the matrix manipulation routines in conjunction with rows and columns generated by AIMMS that can only be indirectly associated with symbolic identifers in the model, AIMMS provides the following suffices which allow you to do so:

- `.ExtendedVariable`, and
- `.ExtendedConstraint`.

These suffices are supported for `Variables`, `Constraints` and `MathematicalPrograms`. They behave like variables and constraints, which implies that it is possible to refer to the `.ReducedCost` and `.ShadowPrice` suffices of these extended suffices to get hold of their sensitivity information.

Suffix dimensions

Each of the suffices listed above has one additional dimension compared to the dimension of the original identifier, over the predefined set `AllGMPExtensions`. For example, assuming that `ae` is an index into the set `AllGMPExtensions`,

- if `z(i,j)` is a variable or constrain, the `.ExtendedVariable` suffix will have indices `z.ExtendedVariable(ae,i,j)`,
- if `mp` is a mathematical program, the `.ExtendedConstraint` suffix will have indices `mp.ExtendedConstraint(ae)`.

Each of the procedures listed above, will add elements to the set `AllGMPExtensions` as necessary. The names of the precise elements added to the set is explained below in more detail.

Suffices generated by `CreateDual`

The procedure `GMP::Instance::CreateDual` will add the following elements to the set `AllGMPExtensions`:

- `DualObjective`, `DualDefinition`, `DualUpperBound`, `DualLowerBound`.

In addition, it will generate the following extended variables and constraints

- For the mathematical program `mp` at hand
  - the variable `mp.ExtendedVariable('DualDefinition')`,
  - the constraint `mp.ExtendedConstraint('DualObjective')`.
- For every ranged constraint `c(i)`
  - the constraint `c.ExtendedConstraint('DualLowerBound',i)`,
  - the constraint `c.ExtendedConstraint('DualUpperBound' i)`.
- For every bounded variable `x(i)` in $[l_i, u_i]$
  - the constraint `x.ExtendedConstraint('DualLowerBound',i)` (if $l_i \neq 0, -\infty$),
  - the constraint `x.ExtendedConstraint('DualUpperBound' i)` (if $u_i \neq 0, \infty$).

Modifying the dual math program

Using the matrix manipulation procedures, you can modify the matrix or objective associated with a dual mathematical program instance created by calling the procedure `GMP::Instance::CreateDual`. Below you will find how you can access the rows and columns of a dual mathematical program instance created by AIMMS.

Row and column names

For each procedure in the `GMP::Coefficient`, `GMP::Row` and `GMP::Column` namespaces you must refer to a scalar constraint and/or variable reference from your symbolic model. For the dual formulation, you must

- use the symbolic primal constraint name, to refer to the dual shadow price variable associated with that constraint in the dual mathematical program instance, and
- use the symbolic primal variable name, to refer to the dual constraint associated with that variable in the dual mathematical program instance.

In other words, when modifying matrix coefficients, rows or columns the role of the symbolic constraints and variables is interchanged.

Implicitly added variables and constraints

You can refer to the implicitly added variables and constraints in the procedures of the `GMP::Coefficient`, `GMP::Row` and `GMP::Column` namespaces through the `.ExtendedVariable` and `.ExtendedConstraint` suffices described above. After solving the dual math program, AIMMS will store the dual solution in the suffices `.ExtendedVariable.ReducedCost` and `.ExtendedConstraint.ShadowPrice`, respectively.

Extended suffices for linearization

By calling the procedures `GMP::Linearization:Add` or `GMP::Linearization::AddSingle`, AIMMS will add the linearization for a single nonlinear constraint instance, or for all nonlinear constraints from a set of nonlinear constraints to a given math program instance. When doing so, AIMMS will add an element `Linearization`*k* (where *k* is a counter) to the set `AllGMPExtensions`, and will create for each nonlinear constraint `c(i)`

- a constraint `c.ExtendedConstraint('Linearization`*k*`',i)`, and
- a variable `c.ExtendedVariable('Linearization`*k*`',i)` if deviations from the constraint are permitted (see also Section 16.10).

Elimination constraints and variables

By calling the procedure `GMP::Instance::AddIntegerEliminationRows`, AIMMS will add one or more constraints and variables to a math program instance, which will eliminate the current integer solution from the math program instance. When called, AIMMS will add elements of the form

- `Elimination`*k*,
- `EliminationLowerBound`*k*, and
- `EliminationUpperBound`*k*

to the set `AllGMPExtensions`. In addition, AIMMS will add

- a constraint mp.ExtendedConstraint('Linearization*k*') to exclude current solution for all binary variables from the math program mp at hand, and
- for every integer variable c(i) with a level value between its bounds the variables and constraints
  - c.ExtendedVariable('Elimination*k*',i),
  - c.ExtendedVariable('EliminationLowerBound*k*',i),
  - c.ExtendedVariable('EliminationUpperBound*k*',i),
  - c.ExtendedConstraint('Elimination*k*',i),
  - c.ExtendedConstraint('EliminationLowerBound*k*',i), and
  - c.ExtendedConstraint('EliminationUpperBound*k*',i).

16.4 Managing the solution repository

The solution repository

The GMP library maintains a solution repository for every generated mathematical program instance. You can use this repository, for instance, to store

- a number of starting solutions for a NLP problem to be solved successively,
- a number of incumbent solutions as reported by a MIP solver, or
- let a solver store multiple solutions.

If you are using solver sessions to initiate a solver, you must explicitly transfer the initial, intermediate or final solutions between the model, the solution repository and the solver session. As discussed in Section 16.2, the function GMP::Instance::Solve performs these necessary solution transfer steps for you, and uses the fixed solution number 1 for all of its communication.
Some solvers are capable of finding multiple solutions instead of at most one. Examples of such solvers are BARON, CPLEX and GUROBI. When such a solver finds multiple solutions, these solutions are stored in the solution repository from number 1 on upwards. The control mechanism to let solvers find multiple solutions is solver specific:

- BARON 15: For more information see the **Help** file for option Number of best solutions in option category Specific solvers - BARON 15 - General.
- CPLEX 12.7: For more information see the **Help** file for option Do populate in option category Specific solvers - CPLEX 12.7 - MIP solution pool.
- GUROBI 7.0: For more information see the **Help** file for option Pool search mode in option category Specific solvers - GUROBI 7.0 - Solution pool.

Solution repository functions

The procedures and functions of the GMP::Solution namespace are listed in Table 16.6. Through these functions you can

- transfer a solution between the solution repository on the one side and the symbolic model or the solver on the other side,
- obtain and set solution properties of a solution in the repository, or
- perform a feasibility check on a solution in the repository.

`Copy`(*GMP*, *fromSol*, *toSol*)
`Move`(*GMP*, *fromSol*, *toSol*)
`Delete`(*GMP*, *solNo*)
`DeleteAll`(*GMP*)
`GetSolutionsSet`(*GMP*)→`Integers`
`Count`(*GMP*)
`RetrieveFromModel`(*GMP*, *SolNr*)
`SendToModel`(*GMP*, *SolNr*)
`SendToModelSelection`(*GMP*, *SolNr*, *Identifiers*, *Suffices*)
`RetrieveFromSolverSession`(*solverSession*, *SolNr*)
`SendToSolverSession`(*solverSession*, *SolNr*)
`GetObjective`(*GMP*, *SolNr*)
`GetBestBound`(*GMP*, *SolNr*)
`GetProgramStatus`(*GMP*, *SolNr*)→`AllSolutionStatus`
`GetSolverStatus`(*GMP*, *SolNr*)→`AllSolutionStatus`
`GetIterationsUsed`(*GMP*, *SolNr*)
`GetMemoryUsed`(*GMP*, *SolNr*)
`GetTimeUsed`(*GMP*, *SolNr*)
`SetObjective`(*GMP*, *SolNr*, *value*)
`SetProgramStatus`(*GMP*, *SolNr*, *PrStatus*)
`SetSolverStatus`(*GMP*, *SolNr*, *PrStatus*)
`SetIterationCount`(*GMP*, *SolNr*, *IterCnt*)
`GetColumnValue`(*GMP*, *SolNr*, *column*)
`SetColumnValue`(*GMP*, *SolNr*, *column*, *value*)
`GetRowValue`(*GMP*, *SolNr*, *row*)
`SetRowValue`(*GMP*, *SolNr*, *row*, *value*)
`Check`(*GMP*, *SolNr*, *NumInf*, *SumInf*, *MaxInf*[, *skipObj*])
`IsInteger`(*GMP*, *SolNr*)
`IsPrimalDegenerated`(*GMP*, *SolNr*)
`IsDualDegenerated`(*GMP*, *SolNr*)
`GetFirstOrderDerivative`(*GMP*, *SolNr*, *row*, *column*)
`ConstraintListing`(*GMP*, *SolNr*, *name*)

Table 16.6: Procedures and functions in `GMP::Solution` namespace

Solution contents

Each solution in the repository is represented by a solution vector containing all relevant solution data, such as

- solution status,
- level values,
- basis information,
- marginals, and
- other relevant requested sensitivity information.

Solution numbering

Each generated mathematical program instance has its own associated solution repository. Each solution in the repository is represented by an integer solution number. Through the function `GMP::Solution::GetSolutionsSet` you can retrieve a subset of the predefined set `Integers` containing the set of all solution numbers that are currently in use for the given mathematical program instance.

Solution transfer to the model

Through the functions

- `GMP::Solution::RetrieveFromModel`,
- `GMP::Solution::SendToModel`, and
- `GMP::Solution::SendToModelSelection`

you can (re-)initialize a solution with the values currently contained in the symbolic model, and vice versa. The function `SendToModelSelection` allows you to only initialize a part of the model identifiers and suffices with a solution of from the solution repository.

Solution transfer to a solver session

Through the functions

- `GMP::Solution::RetrieveFromSolverSession`, and
- `GMP::Solution::SendToSolverSession`

you can set a solution in the repository equal to a solution reported by a given solver session, or initialize the (initial) solution of a solver session with a solution stored in the repository. Notice that these functions do not have a *GMP* argument. Because each solver session is uniquely associated with a single mathematical program instance, AIMMS is able to determine the correct solution repository.

Computing first order derivatives

Using the function `GMP::Solution::GetFirstOrderDerivative`, you can compute, for the given solution, first order derivative of a particular row in a mathematical program with respect to a given variable. You can use such a function, for instance, to implement a sequential linear programming approach for nonlinear programs, as outlined in Section 16.12.5.

16.5 Using solver sessions

Using solver sessions

The procedures and functions of the `GMP::SolverSession` namespace are listed in Table 16.7. Solver sessions are created implicitly by AIMMS or explicitly by calling the procedure `GMP::Instance::CreateSolverSession`.

Solving a mathematical program instance

By calling the `GMP::SolverSession::Execute` procedure, the given solver session will take care of solving the associated mathematical program instance in a blocking manner, i.e. the function will not return until the solver has completed the solution process. This function is called implicitly by the `GMP::Instance::Solve` function or by the `SOLVE` statement.

`Execute`(*solverSession*) `AsynchronousExecute`(*solverSession*)

`ExecutionStatus`(*solverSession*)→`AllExecutionStatuses`
`Interrupt`(*solverSession*)
`WaitForCompletion`(*Objects*)
`WaitForSingleCompletion`(*Objects*)→`AllSolverSessionCompletionObjects`

`CreateProgressCategory`(*solverSession*[, *Name*][, *Size*])
`GetOptionValue`(*solverSession*, *optionName*) `SetOptionValue`(*solverSession*, *optionName*, *value*)

`GetInstance`(*solverSession*)→`AllGeneratedMathematicalPrograms`
`GetSolver`(*solverSession*)→`AllSolvers`
`GetCallbackInterruptStatus`(*solverSession*)→`AllSolverInterrupts`

`GetIterationsUsed`(*solverSession*) `GetMemoryUsed`(*solverSession*) `GetTimeUsed`(*solverSession*)
`GetBestBound`(*solverSession*) `GetObjective`(*solverSession*)

`GetProgramStatus`(*solverSession*)→`AllSolutionStates`
`GetSolverStatus`(*solverSession*)→`AllSolutionStates`
`SetSolverStatus`(*solverSession*)

`GenerateCut`(*solverSession*, *row*[, *local*][, *purgeable*]) `RejectIncumbent`(*solverSession*)
`Transfer`(*solverSession*, *GMP*)

Table 16.7: Procedures and functions in `GMP::SolverSession` namespace

Asynchronous solve

Alternatively, you can solve a mathematical program instance in an non-blocking manner by using the function `GMP::SolverSession::AsynchronousExecute`. Rather than waiting for the solution process to complete, this function will dispatch the solution process to a separate thread of execution, and return immediately. This allows multiple mathematical program instances to be solved in parallel, assuming your computer has multiple processors or a multi-core processor. Note that requests for a synchronous solve through the `SOLVE` statement will fail if a AIMMS is still executing an asynchronous solution process.

Session synchronization

To allow your application to synchronize its execution when multiple solver sessions are executed asynchronously, AIMMS offers the following synchronization procedures

- `GMP::SolverSession::Interrupt`,
- `GMP::SolverSession::ExecutionStatus`,
- `GMP::SolverSession::WaitForCompletion`, and
- `GMP::SolverSession::WaitForSingleCompletion`.

Through the `GMP::SolverSession::Interrupt` function you can request AIMMS to interrupt a solver session that is executing (asynchronously). You can call the function `GMP::SolverSession::ExecutionStatus` to check the status of a given solver session.

Waiting for multiple completions

Using the function `GMP::SolverSession::WaitForCompletion` you can halt the main AIMMS thread of execution to wait until the entire set of solver sessions passed as an argument to the function have completed. You can use this function, for instance, to end the solution phase of your model, prior to moving on to the post-processing phase of your model.

. . . and for single completion

In addition, AIMMS offers a function `GMP::SolverSession::WaitForSingleCompletion` which returns as soon as a single solver session from the given set of solver sessions has completed its execution. The return value of the function is the completed solver session that caused the function to return. You can use `WaitForSingleCompletion`, for instance, to asynchronously solve the next mathematical program instance from a queue of mathematical program instances waiting to be solved.

No solution transfer

Note that neither `GMP::SolverSession::Execute` and `GMP::SolverSession::AsynchronousExecute` will copy the initial solution into the solver, or copy the final solution back into solution repository or model identifiers. When you use these functions you always have to explicitly call functions from the `GMP::Solution` namespace to accomplish these tasks.

Support for callbacks

When callbacks for the mathematical program instance associated with a solver session have been set (see also Section 16.2), AIMMS will make sure that the specified callback procedures in your model will be called whenever appropriate. If you have specified a single callback procedure for multiple callback reasons, you can call the procedure

- `GMP::SolverSession::GetCallbackInterruptStatus`

to retrieve the reason why your callback procedure was called. The result is an element in the predeclared set `AllSolverInterrupts` which contains the elements

- `Incumbent`,
- `NewIncumbent`,
- `AddCut`,
- `Iterations`,
- `Heuristic`,
- `StatusChange`, and
- `Finished`.

When the solver session has not yet been called, the status is '' (empty element). During a callback, you can call the function

- `GMP::SolverSession::GetInstance`

if you need the mathematical program instance associated with the given solver session, and you can retrieve the current objective values using the functions

- `GMP::SolverSession::GetBestBound`, and
- `GMP::SolverSession::GetObjective`.

Synchronous nested solves allowed

During any callback you are allowed to generate and solve other mathematical program instances *in a synchronous manner*. You can use such nested solves, for instance, for finding a heuristic solution during a `Heuristic` callback. Once you have found a heuristic solution, you can pass it onto the running solver session using the function `GMP::Solution::SendToSolverSession`. Note that this functionality is currently only supported by CPLEX and GUROBI.

No asynchronous solves

During a callback AIMMS does not allow you to call the function `GMP::SolverSession::AsynchronousExecute` to solve another mathematical program instance in an asynchronous manner. However, AIMMS offers a special class of synchronization objects called *events*, which allow you to notify the main thread of execution that some event has occurred and act accordingly. When set during a callback, the main thread of execution may respond, for instance, by generating a mathematical program instance based on solver data set by the callback, and solve that mathematical program instance in an asynchronous manner. Events are discussed in full detail in Section 16.6.

Adding cuts

During an `AddCut` callback you may use the procedure `GMP::SolverSession::GenerateCut` to generate a local or global cut. A local cut will only be added to the current node in the solution process and all its descendant nodes, while a global cut will remain to exist for all nodes onwards. The result of the procedure will be the temporary addition of row to the matrix, as if `GMP::Row::Generate` had been called. Note that this functionality is currently only supported by CPLEX, GUROBI and ODH-CPLEX.

Rejecting incumbents

During an `Incumbent` callback you can reject the incumbent found by the solver by calling the procedure `GMP::SolverSession::RejectIncumbent`. Note that this functionality is currently only supported by CPLEX.

Setting options

You can set options for a specific solver session associated through the function `GMP::SolverSession::SetOptionValue`. These option values override the option values for the associated *GMP*, set through `GMP::Instance::SetOptionValue`, which in their turn override the project options.

16.6 Synchronization events

Events for synchronization

To allow for more advanced thread synchronization during parallel solves, AIMMS offers synchronization objects called *events*, which can be manipulated using the function listed in Table 16.8.

`Create`(*name*)→`AllGMPEvents`
`Delete`(*event*)
`Set`(*event*)
`Reset`(*event*)

Table 16.8: Procedures and functions in `GMP::Event` namespace

Creating events

Through the function `GMP::Event::Create` you can create a new event, while the function `GMP::Event::Delete` deletes existing events. Using the function `GMP::Event::Set` you can notify AIMMS that an event has occurred. The function `GMP::Event::Reset` resets the event.

Waiting for events

Events are elements of the predefined set `AllGMPEvents`, which, along with the set `AllSolverSessions`, is a subset of the predefined set `AllSolverSessionCompletionObjects`. As both the functions

- `GMP::SolverSession::WaitForCompletion`, and
- `GMP::SolverSession::WaitForSingleCompletion`

expect a subset of the set `AllSolverSessionCompletionObjects` as their arguments, these functions can be used to wait for both solver session completion and the occurrence of events.

Using events

You can use events, for example, to notify the main thread of execution in your model that you want a new mathematical program instance to be generated and solved asynchronously based on input data provided by a solver callback. AIMMS does not allow asynchronous solves to be started from within a callback itself.

16.7 Supporting functions for stochastic programs

Supporting functions for stochastic models

The stochastic Benders algorithm (see Section 19.4.2) is implemented in AIMMS as a combination of a system module that can be included into your model, and a number of supporting functions in the `GMP::Stochastic` namespace of the GMP library. The procedures and functions of the `GMP::Stochastic` namespace are listed in Table 16.9.

`BendersFindFeasibilityReference`(*GMP*, *stage*, *scenario*) →`AllGeneratedMathematicalPrograms`
`BendersFindReference`(*GMP*, *stage*, *scenario*) →`AllGeneratedMathematicalPrograms`
`CreateBendersRootproblem`(*GMP*[, *name*]) →`AllGeneratedMathematicalPrograms`
`UpdateBendersSubproblem`(*GMP*, *solution*)
`AddBendersFeasibilityCut`(*GMP*, *solution*, *cutNo*)
`AddBendersOptimalityCut`(*GMP*, *solution*, *cutNo*)
`MergeSolution`(*GMP*, *solution1*, *solution2*[, *updObj*])
`GetRepresentativeScenario`(*GMP*, *stage*, *scenario*)→`AllStochasticScenarios`
`GetObjectiveBound`(*GMP*, *solution*)
`GetRelativeWeight`(*GMP*, *stage*, *scenario*)

Table 16.9: Procedures and functions in `GMP::Stochastic` namespace

Overview of functionality

For a more detailed overview of the functionality offered by the functions in the `GMP::Stochastic` namespace, we refer to

- Section 19.4.2 for an outline of the stochastic Benders algorithm,
- the system module containing the AIMMS implementation of the stochastic Benders algorithm, and
- the AIMMS Function Reference for a detailed explanation of the functionality of each function.

16.8 Supporting functions for robust optimization models

Supporting functions for robust models

Table 16.10 lists the functions available in the `GMP::Robust` namespace in support of working with robust optimization models.

`EvaluateAdjustableVariables`(*GMP*, *Variables*[, *merge*])

Table 16.10: Procedures and functions in `GMP::Robust` namespace

Overview of functionality

For a more detailed overview of the functionality offered by the functions in the `GMP::Robust` namespace, we refer to

- Section 20.4 for an outline of the functionality offered by the procedure `GMP::Robust::EvaluateAdjustableVariables`, and
- the AIMMS Function Reference for a more detailed explanation.

16.9 Supporting functions for Benders' decomposition

Supporting functions for Benders' decomposition

The Benders' decomposition algorithm (see Chapter 21) is implemented in AIMMS as a combination of a system module that can be included into your model, and a number of supporting functions in the `GMP::Benders` namespace of the GMP library. The procedures and functions of the `GMP::Benders` namespace are listed in Table 16.11.

`CreateMasterProblem`(*GMP*, *Variables*, *name*[, *feasibilityOnly*][, *addConstraints*]) →`AllGeneratedMathematicalPrograms`
`CreateSubProblem`(*GMP1*, *GMP2*, *name*[, *useDual*][, *normalizationType*]) →`AllGeneratedMathematicalPrograms`
`UpdateSubProblem`(*GMP1*, *GMP2*, *solution*[, *round*])
`AddFeasibilityCut`(*GMP1*, *GMP2*, *solution*, *cutNo*)
`AddOptimalityCut`(*GMP1*, *GMP2*, *solution*, *cutNo*)

Table 16.11: Procedures and functions in `GMP::Benders` namespace

Overview of functionality

For a more detailed overview of the functionality offered by the functions in the `GMP::Benders` namespace, we refer to

- Chapter 21 for an outline of the Benders' decomposition algorithm,
- the system module containing the AIMMS implementation of the Benders' decomposition algorithm, and
- the AIMMS Function Reference for a detailed explanation of the functionality of each function.

16.10 Creating and managing linearizations

MINLP problems and linearizations

When solving a mixed integer nonlinear (MINLP) problem using an outer approximation approach (see also Chapter 18 for a more detailed description), an associated master MIP problem is created and extended with linearizations of the nonlinear constraints of the original problem with respect to successive solutions of the underlying NLP sub-problem. Using the procedures in the `GMP::Linearization` namespace, AIMMS allows you to add linearizations of nonlinear constraints to a particular math program instance. Together with the `GMP::Instance::CreateMasterMIP` procedure to create the initial master MIP problem, these procedures form the heart of the implementation of the outer approximation algorithm in AIMMS, as discussed in Section 18.6.

Managing linearizations

The procedures and functions of the `GMP::Linearization` namespace are listed in Table 16.12.

`Add`(*GMP1*, *GMP2*, *solNr*, *conSet*, *devPermitted*, *penalty*, *linNr*[, *jacTol*])
`AddSingle`(*GMP1*, *GMP2*, *solNr*, *row*, *devPermitted*, *penalty*, *linNr*[, *jacTol*])
`Delete`(*GMP*, *linNr*)
`RemoveDeviation`(*GMP*, *row*, *linNr*)
`GetDeviation`(*GMP*, *row*, *linNr*)
`GetDeviationBound`(*GMP*, *row*, *linNr*)
`GetWeight`(*GMP*, *row*, *linNr*)
`GetLagrangeMultiplier`(*GMP*, *row*, *linNr*)
`GetType`(*GMP*, *row*, *linNr*)→`AllRowTypes`
`SetDeviationBound`(*GMP*, *row*, *linNr*, *value*)
`SetWeight`(*GMP*, *row*, *linNr*, *value*)
`SetType`(*GMP*, *row*, *linNr*, *rowType*)

Table 16.12: Procedures and functions in `GMP::Linearization` namespace

Creating and deleting linearizations

Through the procedures

- `GMP::Linearization::Add`,
- `GMP::Linearization::AddSingle`,
- `GMP::Linearization::Delete`, and
- `GMP::Linearization::RemoveDeviation`

you can instruct AIMMS to add and delete one or more rows and columns to a given math program instance, representing the linearizations of (nonlinear) constraints of another math program instance at a particular solution point.

Modifying linearizations

You can modify the rows and columns generated by these procedures using the matrix manipulation routines discussed in Section 16.3. The rows and columns generated by AIMMS cannot be associated directly with constraints and variables in your model, but must be addressed using the `.ExtendedConstraint` and `.ExtendedVariable` suffices. Section 16.3.6 discusses the precise suffices generated by AIMMS when using the functions `GMP::Linearization::Add` and `GMP::Linearization::AddSingle`.

Remaining functions

Through the remaining functions in the `GMP::Linearization` namespace you can

- get and set information about the devation variables added to the linearized constraints, and their penalties added to the objective, and
- get and set the row types of the generated constraints.

Note the you must use the appropriate `.ExtendedConstraint` suffix to refer to the particular linearization constraint when using these functions.

16.11 Customizing the progress window

Customizing the progress window

When you are using the GMP library to implement a customized algorithm for a particular problem or problem class, you can use the procedures in the `GMP::ProgressWindow` namespace to customize the contents of the AIMMS Progress Window. This allows you to provide customized feedback to the end-user regarding the progress of the overall solution algorithm, or to provide simultaneous progress information about multiple solver session executing in parallel.

Customizing progress

The procedures and functions of the `GMP::ProgressWindow` namespace are listed in Table 16.13. They allow you to modify every aspect of the solver part of the AIMMS progress window.

`DisplaySolver`(*name*[, *Category*])
`DisplayLine`(*lineNr*, *title*, *value*[, *Category*])
`DisplayProgramStatus`(*status*[, *Category*][, *lineNo*])
`DisplaySolverStatus`(*status*[, *Category*][, *lineNo*])
`FreezeLine`(*lineNo*, *totalFreeze*[, *Category*])
`UnfreezeLine`(*lineNo*[, *Category*])
`DeleteCategory`(*Category*)
`Transfer`(*Category*, *solverSession*)

Table 16.13: Procedures and functions in `GMP::ProgressWindow` namespace

Creating a new progress category

When your model executes multiple solver sessions in parallel, you can request AIMMS to create a new progress *category* to display separate solver progress for each solver session in a separate area of the progress window. Using the function `GMP::Instance::CreateProgressCategory`, you can create a new progress category for a specific mathematical program instance that will subsequently be used to display solver progress for *all* solver sessions associated with that mathematical program instance. Alternatively, you can create a per-session category to display separate solver progress for every single solver session using the function `GMP::SolverSession::CreateProgressCategory`. The procedure `GMP::ProgressWindow::Transfer` allows you to share a progress category among several solver sessions. Through the function `GMP::ProgressWindow::DeleteCategory` you can delete progress categories created by either function.

Freezing category content

Through the functions `GMP::ProgressWindow::FreezeLine` and `GMP::ProgressWindow::UnfreezeLine` you can instruct AIMMS to stop and start updating particular areas of the solver progress area associated with the progress category.

Displaying custom content

When you are writing a custom algorithm you can use the progress window to display custom progress information supplied by you using the functions

- `GMP::ProgressWindow::DisplaySolver`,
- `GMP::ProgressWindow::DisplayLine`,
- `GMP::ProgressWindow::DisplayProgramStatus`, and
- `GMP::ProgressWindow::DisplaySolverStatus`.

When your custom algorithm consists of a sequence of solves, you can use these functions, for instance, to display custom progress information for the overall algorithm, possibly in combination with regular progress for the underlying solves in a separate category.

Example of use

An example of the usage of the `GMP::ProgressWindow` can be found in the AIMMS module containing the GMP Outer Approximation algorithm discussed in Section 18.6. In this module, the contents of the AIMMS progress window is adapted for the AIMMS Outer Approximation solver.

16.12 Examples of use

This section

In this section there are five examples to illustrate the use of the GMP library. Each example consists of two paragraphs. The first paragraph explains the basic problem and an algorithmic approach, while the second paragraph provides the corresponding implementation in AIMMS using the GMP procedures. Note that these algorithms could also have been implemented using AIMMS' regular `SOLVE` statement, but at the cost of one or more structure recognition steps during every iteration.

16.12.1 Indexed mathematical program instances

Indexed mathematical program instances

AIMMS does not support indexed mathematical program declarations, which would result in a different mathematical program for every index value when generated. Using the GMP library, however, it is straightforward to generate indexed mathematical program *instances*

Declarations in AIMMS

Consider the following declarations

```
Set Cities {
    Index       : c, j;
}
Set  SelectedCities {
    SubsetOf    : Cities;
    Index       : i;
}
```

together with a mathematical program declaration `TransportModel` defining a standard transportation problem determining transports from cities `i` to `j`.

When SelectedCities equals Cities we would solve the mathematical program for all possible combinations of cities.

Procedure in AIMMS

Further assume that we have an element parameter IndexedTransportModel(c) into the set AllGeneratedMathematicalPrograms. The following procedure illustrates how indexed mathematical program instances for every city c which restricts the transports from c to all cities j.

```
for ( c ) do
    SelectedCities := {c};

    IndexedTransportModel(c) := GMP::Instance::Generate( TransportModel,
                         "TransportModel-" + FormatString("%e", c) );
endfor;
```

16.12.2 Sensitivity analysis

Parametric changes

Sensitivity analysis considers how the optimal solution, and the corresponding objective function value, change as a result of changes in input data. Using the GMP library, it is straightforward to write a procedure to determine these sensitivities for a discrete set of input values.

Procedure in AIMMS

The following procedure illustrates how parametric changes can be implemented using matrix manipulation functions. The resulting objective function values are stored in a separate identifier.

```
myGMP := GMP::Instance::Generate( MathProgramOfInterest );

for ( n ) do
   GMP::Coefficient::Set( myGMP,
                          ResourceConstraint(SelectedResource),
                          ActivityVariable(SelectedActivity),
                          OriginalCoefficient + Delta(n) );

   GMP::Instance::Solve( myGMP );

   ObjectiveValue(n) := GMP::Solution::GetObjective(myGMP, 1);
endfor;
```

16.12.3 Finding a feasible solution for a binary program

Fixing one variable at a time

There have been instances in which the following simple but greedy heuristic was used successfully to solve a binary program. The algorithm considers linear programming solutions in sequence. During each iteration, the algorithm

- selects the single variable that, of all the variables, is nearest but not equal to one of its bounds, and
- fixes the value of this variable to that of the nearest bound.

As soon as such variables can no longer be found (and the last linear programming solution is optimal), a feasible integer solution to the binary program has been found.

Procedure in AIMMS

The following procedure illustrates how fixing one variable at a time can be implemented using matrix manipulation functions. The procedure terminates as soon as there is no solution, or all variables have been fixed.

```
relaxedGMP := GMP::Instance::Generate( RelaxedBinaryProgram );
GMP::Instance::Solve( relaxedGMP );

repeat
    LargestLessThanOne      := ArgMax( j | x(j) <= 1 - Tolerance, x(j) );
    SmallestGreaterThanZero := ArgMin( j | x(j) >= Tolerance,     x(j) );

    break when ( RelaxedBinaryProgram.ProgramStatus = 'Infeasible' or
                 not ( LargestLessThanOne or SmallestGreaterThanZero ) );

    if ( x(SmallestGreaterThanZero) < 1 - x(LargestLessThanOne) )
    then GMP::Column::Freeze( relaxedGMP, x(SmallestGreaterThanZero), 0 );
    else GMP::Column::Freeze( relaxedGMP, x(LargestLessThanOne), 1 );
    endif;

    GMP::Instance::Solve( relaxedGMP );
endrepeat;
```

16.12.4 Column generation

Adding columns

Chapter 20 of the AIMMS book on Optimization Modeling describes a cutting stock problem. This problem is modeled as a linear program with an initial selection of cutting patterns. An auxiliary integer programming model is introduced to generate a new "best" pattern based on the current solution of the linear program and the corresponding shadow prices. Such a pattern is then added to the existing patterns in the linear program, and the next optimal solution is found. This process continues until no further improvement in the value of the objective function can be achieved.

Procedure in AIMMS

The following procedure illustrates how adding columns can be implemented using matrix manipulation functions. During each iteration of the overall process, two different mathematical programs are modified in turn.

```
cuttingStockGMP := GMP::Instance::Generate( CuttingStock );
GMP::Instance::Solve( cuttingStockGMP );

findPatternGMP := GMP::Instance::Generate( FindPattern );
GMP::Instance::Solve( findPatternGMP );

MaxPattern := 0;
while ( PatternContribution > 1 ) do
    MaxPattern += 1;
    AllPatterns += MaxPattern;
    LastPattern := last(AllPatterns);
```

```
        GMP::Column::Add( GMP: cuttingStockGMP, column: RollsUsed(LastPattern) );

        for ( width ) do
            GMP::Coefficient::Set( GMP   : cuttingStockGMP,
                                   row   : MeetCutDemand(width),
                                   column: RollsUsed(LastPattern),
                                   value : CutsInPattern(width)  );
        endfor;
        GMP::Instance::Solve( cuttingStockGMP );

        for ( width ) do
            GMP::Coefficient::Set( GMP   : findPatternGMP,
                                   row   : PatternContribution,
                                   column: CutsInPattern(width),
                                   value : MeetCutDemand(width).ShadowPrice );
        endfor;
        GMP::Instance::Solve( findPatternGMP );
    endwhile;
```

Here `MaxPattern` is a parameter of type integer, `AllPatterns` a subset of `Integers`, and `LastPattern` an element parameter with range `AllPatterns`.

16.12.5 Sequential linear programming

Sequential linear programming

Linear constraints and a nonlinear objective function together form a special class of nonlinear programs. It is possible to solve a problem of this class by solving a sequence of linear programs. The main requirement is that the nonlinear objective function has first-order derivatives. The objective function can then be linearized around the solution of a previous linear program. By restricting the linearized function to an appropriate finite box, a new solution point is found. The sequence of linear programs terminates when the appropriate box has become sufficiently small. Upon termination, the optimal solution, as last found, is considered to be a local optimum of the underlying nonlinear program.

Procedure in Aimms

The following procedure illustrates how sequential linear programming can be implemented using matrix manipulation functions. The procedure assumes the existence of finite upper and lower bounds on the variables, and the presence of a function `ComputeGradient` to compute the required first partial derivatives with respect to the variables in the objective function. To implement the function `ComputeGradient` one can, for instance, use the built-in GMP function `GMP::Solution::GetFirstOrderDerivative` (see also Section 16.4).

```
    linearizedGMP := GMP::Instance::Generate(LinearizedProgram);
    GMP::Instance::Solve(linearizedGMP);

    BoxWidth(j)  := 0.1 * (x.upper(j) - x.lower(j));
    x(j)         := 0.5 * (x.upper(j) + x.lower(j));

    while ( max( j, BoxWidth(j) ) > Tolerance ) do
       ObjCoeff(j) := ComputeGradient(x)(j);
```

```
   for (j) do
      GMP::Column::SetLowerBound ( linearizedGMP, x(j),
                                   max(x.lower(j), x(j) - 0.5*BoxWidth(j)) );
      GMP::Column::SetUpperBound ( linearizedGMP, x(j),
                                   min(x.upper(j), x(j) + 0.5*BoxWidth(j)) );
      GMP::Coefficient::Set( linearizedGMP, ObjectiveRow,
                             x(j), ObjCoeff(j)            );
   endfor;
   GMP::Instance::Solve(linearizedGMP);

   BoxWidth(j) *= ShrinkFactor;
endwhile;
```

Bibliography

[Ch04] J.W. Chinneck, *The constraint consensus method for finding approximately feasible points in nonlinear programs*, INFORMS Journal on Computing **16** (2004), 255–265.