Gurobify

A GAP interface to Gurobi.

0.1

23/02/2017

Jesse Lansdown

Jesse Lansdown
Email: jesse.lansdown@research.uwa.edu.au
Homepage: www.jesselansdown.com
Address: Jesse Lansdown
Lehrstuhl B für Mathematik
RWTH Aachen
Pontdriesch 10 - 16
52062 Aachen
Germany

Abstract

Gurobify provides an interface to the Gurobi Optimizer software from GAP. It enables the creation and modification of mixed integer and linear programmming models which can be solved directly by Gurobi from within the GAP environment.

Copyright

Acknowledgements

I thank Sebastian Gutsche for generously taking the time to explain the inner workings of GAP and GAP packages to me, and for pointing me towards examples. I also thank John Bamberg for introducing me to both GAP and Gurobi and showing me how they can be used to so effectively complement each other. I used the AutoDoc[GH16] package to streamline the creation of the documentation for this package, and PackageMaker[Hor16] to generate a package template. I would also like to acknowledge the support of an Australian Government Research Training Program (RTP) Scholarship while writing this software.

Contents

Chapter 1

Introduction

1.1 What is Gurobify?

Gurobify is a GAP[GAP16] package which provides an interface to the optimisation software Gurobi[gur16a]. Please use the issue tracker to inform me of any bugs or suggestions. I would also like to hear about applications of Gurobify via email.

1.2 Citing Gurobify

I am interested to know who is using Gurobify! If you obtain a copy I would appreciate it if you sent me an email to let me know. If Gurobify aids you in obtaining results that lead to a publication, please cite Gurobify as you would a paper. An example BibTeX entry for citing gurobify is given below. Please also send me an email informing me of the paper for my own interest.

Example
```
@manual{gurobify,
Author = {Lansdown, Jesse},
Key = {gurobify},
Title = {{Gurobify -- A GAP interface to Gurobi, Version 1.0}},
Url = {\verb+(https://github.com/jesselansdown/Gurobify/)+},
Year = 2017}
```

Here is the entry in the APA style which may be used directly in the bibliography environment of your LaTeX document.

Example
```
\bibitem[Gurobify]{gurobify}
J.~Lansdown.
\newblock Gurobify -- A {GAP} interface to Gurobi, version 1.0, 2017.
```

1.3 Prerequisites

Gurobi requires the following software to be installed.

- GAP 4.8 (or later)
- Gurobi 7.0

- Autotools

Autotools may be installed on MacOSX using homebrew with the commands *brew install autoconf* and *brew install automake*. If you want to regenerate the documentation for any reason, then the following will also be required.

- AutoDoc 2016.03.08 (or later)
- GAPDoc 1.5 (or later)

Although Gurobi is proprietary software, it is available free for academic use. According to the Gurobi website,

"Gurobi makes most of our solvers available to recognized degree-granting academic institutions free of charge" [gur16c],

and

"The free Academic License for Gurobi has all the features and performance of the full Gurobi Optimizer. A free Academic License has no limits on model size. The only restrictions on the use of these licenses are:

- They can only be used by faculty, students, or staff of a degree-granting academic institution
- They can only be used for research or educational purposes
- They must be validated from a recognized academic domain, as described below.

Note, free academic licenses expire twelve (12) months after the date on which your license was generated, but eligible faculty, students, or staff can renew a license by repeating the above process" [gur16b].

For up-to-date information on Gurobi licences, please refer to the Gurobi website. A link can be found in the Appendix, or through a simple search online.

1.4 Installation

To install Gurobify, first unpack it in the pkg directory of the GAP installation directory. You may place Gurobify in a different location, so long as its parent directory is called "pkg". Installing Gurobify outside of the GAP root pkg directory is not recommended, but is useful for example when administrative privelages are needed to access the GAP root directory. Next run the following commands in the terminal from within the Gurobify directory.

Example
```
./autogen.sh
./congigure --with-gurobi=<gurobi path> [--with-gaproot=<gap path>]
make
```

The --with-gurobi=<gurobi path> is a necessary argument, and normally looks something like this on MacOSX,

--with-gurobi=/Library/gurobi701/mac64

or something like this on Linux,

--with-gurobi=/opt/gurobi701/linux64

The [--with-gaproot=<gap path>] is an optional argument, and is not normally necessary if Gurobify is placed in the pkg directory of the GAP root directory. If, however, Gurobify is located in a non-root pkg directory, then this argument must be included. It normally looks something like this,

--with-gaproot=/opt/gap4r8/

1.5 Documentation

Within the Gurobify directory there is a subdirectory called doc. This directory contains the documentaion for Gurobify in the form of a pdf file called "manual.pdf" as well as in html form. To access the html version of the manual, open the file called "chap0.html". The documentation can be regenerated by running the following command in the terminal from the Gurobify directory, though this should not be necessary.

Example
```
gap.sh makedoc.g
```

1.6 Loading Gurobify

Open GAP and load Gurobify with the command *LoadPackage("Gurobify");*. You should see something like the following.

Example
```
gap>  LoadPackage("Gurobify");
-----------------------------------------------------------------------------
Loading  Gurobify 1.0 (Gurobify provides an interface to Gurobi from GAP.)
by Jesse Lansdown (www.jesselansdown.com).
Homepage: https://github.com/jesselansdown/Gurobify/
-----------------------------------------------------------------------------
true
```

Note that if you have Gurobify located somewhere other than the GAP root directory's pkg directory, then you must run GAP with the following command to enable GAP to find Gurobify.

Example
```
gap.sh -l ";/alternative/path/to/Gurobify"
```

where */alternative/path/to/Gurobify* is the path to the directory which contains */pkg/Gurobify* as subdirectories.

Chapter 2

Getting Started

2.1 Getting Started

TODO

2.2 Minimal working example?

TODO

Example

```
gap> model := GurobiNewModel(["BINARY", "BINARY", "BINARY"], [1.,2.,1.]);
<object>
gap> GurobiSetIntegerAttribute(model, "ModelSense", -1);
gap> GurobiAddConstraint(model, [2, 2, 2], "<", 6, "Initial Constraint");
gap> GurobiAddConstraint(model, [1, 2, 3], ">", 5, "Initial Constraint");
gap> GurobiOptimizeModel(model);
2
gap> GetSolution(model);
[ 1., 1., 1. ]
gap> GurobiReset(model);
gap> GurobiSetIntegerAttribute(model, "ModelSense", 1);
gap> GurobiOptimizeModel(model);
2
gap> GetSolution(model);
[ 0., 1., 1. ]
gap> GurobiWriteToFile(model, "test.lp");
gap> re_model := GurobiReadModel("test.lp");
<object>
gap> GurobiAddConstraint(re_model, [1, 1, 1], ">", 3, "Other Constraint");
gap> GurobiOptimizeModel(re_model);
2
gap> GetSolution(re_model);
[ 1., 1., 1. ]
gap> GurobiAddConstraint(re_model, [0, 1, 1], "<", 1, "Other Constraint");
gap> GurobiOptimizeModel(re_model);
3
gap> GurobiDeleteAllConstraintsWithName(re_model, "Other Constraint");
gap> GurobiOptimizeModel(re_model);
2
```

```
gap> GetSolution(re_model);
[ 0., 1., 1. ]
gap> SetTimeLimit(re_model, 0.0001);
true
gap> GurobiOptimizeModel(re_model);
9
gap> SetTimeLimit(re_model, 0.01);
true
gap> GurobiOptimizeModel(re_model);
2
```

Chapter 3

Using Gurobify

3.1 Creating or reading a model

TODO intro to section

3.1.1 GurobiReadModel

▷ `GurobiReadModel(`*`ModelFile`*`)` (function)

Returns: a Gurobi model.

Takes a model file, reads it and creates a Gurobi model from it. ModelFile is the name of the file as a string, with the appropriate extension, and including the path if the file is not located in the current GAP working directory. Gurobi accepts files of type .mps, .rew, .lp, .rlp, .ilp, or .opb. Refer to the gurobi documentation for more infomation on which file types can be read.

3.1.2 GurobiSetVariableNames

▷ `GurobiSetVariableNames(`*`Model, VariableNames`*`)` (function)

Returns:

To do: check that everything is a string

3.1.3 GurobiNewModel (for IsList)

▷ `GurobiNewModel(`*`VariableTypes`*`)` (operation)

▷ `GurobiNewModel(`*`VariableTypes[, VariableNames]`*`)` (operation)

Returns: A Gurobi model

Creates a gurobi model with variables defined by VariableTypes and an objective function given by ObjectiveFunction. VariableTypes must be a list, with entries indexed by the set of variables, and entries corresponding to the type of variable, as a string. Accepted variable types are "CONTINUOUS", "BINARY", "INTEGER", "SEMICONT", or "SEMIINT". Refer to the Gurobi documentation for more information on the variable types. ObjectiveFunction is a list, with entries indexed by the set of variables, where each entry corresponds to the coefficient of the variable in the objective function. Optionally takes the names of the variables

3.2 Adding and deleting constraints

TODO

3.2.1 GurobiDeleteAllConstraintsWithName

▷ `GurobiDeleteAllConstraintsWithName(`*`Model, ConstraintName`*`)` (function)

Returns:

Deletes all constraints from a model with the name ConstraintName. Returns the updated model.

3.2.2 GurobiAddConstraint (for IsGurobiModel, IsList, IsString, IsFloat, IsString)

▷ `GurobiAddConstraint(`*`Model, CstrEquation, CstrSense, CstrRHSValue, CstrName`*`)` (operation)

▷ `GurobiAddConstraint(`*`Model, CstrEquation, CstrSense, CstrRHSValue, CstrName`*`)` (operation)

▷ `GurobiAddConstraint(`*`Model, CstrEquation, CstrSense, CstrRHSValue`*`)` (operation)

▷ `GurobiAddConstraint(`*`Model, CstrEquation, CstrSense, CstrRHSValue`*`)` (operation)

Returns: true

Adds a constraint to a gurobi model. CstrEquation must be a list, with entries indexed by the variable set, such that each entry is the coefficient of the corresponding variable in the constraint equation. The CstrSense must be one of "<", ">" or "=", where Gurobi interprets < as <= and > as >=. The CstrRHSValue is the value on the right hand side of the constraint. A constraint may also be given a name, which helps to identify the constraint if it is to be deleted at some point. Note that a model must be updated or optimised before any additional constraints become effective. In the second instance, CstrRHSValue takes an integer Note that the name is an optional argument. It is necessary if you wish to delete the constraint at a later stage. If no name is given, the constraint will be named "UnNamedConstraint".

3.2.3 GurobiAddMultipleConstraints (for IsGurobiModel, IsList, IsList, IsList, IsList)

▷ `GurobiAddMultipleConstraints(`*`Model, ConstraintEquations, ConstraintSenses, ConstraintRHSValues, ConstraintNames`*`)` (operation)

Returns: true

Add a multiple constraints to a model. ConstraintEquations, ConstraintSenses, ConstraintRHSValues and ConstraintNames are lists, such that the i-th entries of each of them determine a single constraint in the same manner as for the operation GurobiAddConstraint.

3.3 Adding and modifying objective functions

3.3.1 GurobiMaximiseModel (for IsGurobiModel)

▷ `GurobiMaximiseModel(`*`Model`*`)` (operation)

Returns: true

Sets the model sense to maximise. When the model is optimized, it will try to maximise the objective function.

3.3.2 GurobiMinimiseModel (for IsGurobiModel)

▷ `GurobiMinimiseModel(`*`Model`*`)` (operation)

Returns: true

Sets the model sense to minimise. When the model is optimized, it will try to minimise the objective function.

3.3.3 GurobiSetObjectiveFunction (for IsGurobiModel, IsList)

▷ `GurobiSetObjectiveFunction(`*`Model, ObjectiveValues`*`)` (operation)

Returns: true

Set the objective function for a model.

3.3.4 GurobiObjectiveFunction (for IsGurobiModel)

▷ `GurobiObjectiveFunction(`*`Model`*`)` (operation)

Returns: List of coefficients of the objective function

View the objectivive function.

3.4 Optimizing a model

TODO

3.4.1 GurobiOptimizeModel

▷ `GurobiOptimizeModel(`*`Model`*`)` (function)

Returns: Optimisation status.

Takes a Gurobi model and optimises it. Returns the optimisation status code which indicates the outcome of the optimisation. A status code of 2 indicates that a feasible solution was found, a status code of 3 indicates the model is infeasible. There a number of other status codes. Refer to the Gurobi documentation for more information about status codes. The model itself is altered to reflect the optimisation, and more information about can be obatained using other functions, in particular the GurobiGetAttribute and GurobiGetAttributeArray functions.

3.4.2 GurobiReset

▷ `GurobiReset(`*`Model`*`)` (function)

Returns:

Reset all information associated with a solution for the model.

3.4.3 GurobiSolution (for IsGurobiModel)

▷ `GurobiSolution(`*`Model`*`)` (operation)

Returns: Solution

Display the solution found for a successfuly optimised model.

3.5 Querying attributes and parameters

Note that a model must be updated or optimized before parameters and attributes are updated, any queries will return the values at the time the model was last updated or optimized.

3.5.1 GurobiNumberOfVariables (for IsGurobiModel)

▷ `GurobiNumberOfVariables(`*`Model`*`)` (operation)

Returns: Number of variables

Returns the number of variables in the model.

3.5.2 GurobiNumberOfConstraints (for IsGurobiModel)

▷ `GurobiNumberOfConstraints(`*`Model`*`)` (operation)

Returns: Number of linear constraints

Returns the number of linear constraints in the model.

3.5.3 GurobiObjectiveValue (for IsGurobiModel)

▷ `GurobiObjectiveValue(`*`Model`*`)` (operation)

Returns: objective value

Returns the objective value of the current solution.

3.5.4 GurobiObjectiveBound (for IsGurobiModel)

▷ `GurobiObjectiveBound(`*`Model`*`)` (operation)

Returns: objective bound

Returns the best known bound for the model.

3.5.5 GurobiRunTime (for IsGurobiModel)

▷ `GurobiRunTime(`*`Model`*`)` (operation)

Returns: objective bound

Returns the wall clock runtime in seconds for the most recent optimization.

3.5.6 GurobiStatus (for IsGurobiModel)

▷ `GurobiStatus(`*`Model`*`)` (operation)

Returns: objective bound

Returns the optimisation status of the most recent optimization. Refer to the Gurobi documentation for more on the optimization statuses. See the appendix for a link.

3.5.7 GurobiNumericFocus (for IsGurobiModel)

▷ `GurobiNumericFocus(`*`Model`*`)` (operation)

Returns: numeric focus

Returns the numeric focus value of the model.

3.5.8 GurobiTimeLimit (for IsGurobiModel)

▷ `GurobiTimeLimit(`*`Model`*`)` (operation)

Returns: time limit

Returns the time limit for the model.

3.5.9 GurobiCutOff (for IsGurobiModel)

▷ `GurobiCutOff(`*`Model`*`)` (operation)

Returns: cutoff value

Returns the cutoff value for the model.

3.5.10 GurobiBestObjectiveBoundStop (for IsGurobiModel)

▷ `GurobiBestObjectiveBoundStop(`*`Model`*`)` (operation)

Returns: best objective bound limit value

Returns the best objective bound limit value for the model.

3.5.11 GurobiMIPFocus (for IsGurobiModel)

▷ `GurobiMIPFocus(`*`Model`*`)` (operation)

Returns: MIP focus

Returns the MIP focus value for the model.

3.5.12 GurobiBestBoundStop (for IsGurobiModel)

▷ `GurobiBestBoundStop(`*`Model`*`)` (operation)

Returns: Best bound stopping value

Returns the best bound stopping value for the model.

3.5.13 GurobiSolutionLimit (for IsGurobiModel)

▷ `GurobiSolutionLimit(`*`Model`*`)` (operation)

Returns: solution limit value

Returns the solution limit value for the model.

3.5.14 GurobiIterationLimit (for IsGurobiModel)

▷ `GurobiIterationLimit(`*`Model`*`)` (operation)

Returns: Iteration limit

Returns the iteration limit value for the model.

3.5.15 GurobiNodeLimit (for IsGurobiModel)

▷ `GurobiNodeLimit(`*`Model`*`)` (operation)

Returns: Node limit

Returns the node limit value for the model.

3.5.16 GurobiVariableNames (for IsGurobiModel)

▷ `GurobiVariableNames(`*`Model`*`)` (operation)

Returns:

TODO

3.6 Querying other attributes and parameters

In addition to these specific queries given in the previous section, all other gurobi parameters and attributes which take integer or double values can be queried using GurobiGetIntegerParameter("ParameterName"), GurobiGetDoubleParameter("ParameterName"), GurobiGetIntegerAttribute("AttributeName") or GurobiGetDoubleAttribute("AttributeName") respectively, where "ParameterName" and "AttributeName" are strings given exactly as in the Gurobi documentation. See the Appendix for links to the relevant documentation.

3.6.1 GurobiGetIntegerParameter

▷ `GurobiGetIntegerParameter(`*`Model, ParameterName`*`)` (function)

Returns: parameter value

Takes a Gurobi model and retrieve the value of a integer-valued parameter. Refer to the Gurobi documentation for a list of parameters and their types.

3.6.2 GurobiGetDoubleParameter

▷ `GurobiGetDoubleParameter(`*`Model, ParameterName`*`)` (function)

Returns: parameter value

Takes a Gurobi model and retrieve the value of a double-valued parameter. Refer to the Gurobi documentation for a list of parameters and their types.

3.6.3 GurobiIntegerAttribute

▷ `GurobiIntegerAttribute(`*`Model, AttributeName`*`)` (function)

Returns: attibute value

Takes a Gurobi model and retrieve the value of an integer-valued attribute. Refer to the Gurobi documentation for a list of attributes and their types.

3.6.4 GurobiDoubleAttribute

▷ `GurobiDoubleAttribute(`*`Model, AttributeName`*`)` (function)

Returns: attibute value

Takes a Gurobi model and retrieve the value of a double-valued attribute. Refer to the Gurobi documentation for a list of attributes and their types.

3.6.5 GurobiAttributeArray

▷ `GurobiAttributeArray(`*`Model, AttributeName`*`)` (function)

Returns: attibute array

Takes a Gurobi model and retrieve an attribute array. Can only get value of attributes arrays which take integer or double values, Refer to the Gurobi documentation for a list of attributes and their types.

3.6.6 GurobiStringAttributeArray

▷ `GurobiStringAttributeArray(`*`Model, AttributeName`*`)` (function)

Returns: attibute array

TODO

3.7 Modifying attributes and parameters

Note that a model must be updated or optimized before parameters and attributes are updated, any queries will return the values at the time the model was last updated or optimized.

3.7.1 GurobiSetTimeLimit (for IsGurobiModel, IsFloat)

▷ `GurobiSetTimeLimit(`*`Model, TimeLimit`*`)` (operation)

Returns: true

Set a time limit for a Gurobi model. Note that TimeLimit should be a float, however an integer value can be given which will be automatically converted to a float.

3.7.2 GurobiSetBestObjectiveBoundStop (for IsGurobiModel, IsFloat)

▷ `GurobiSetBestObjectiveBoundStop(`*`Model, BestObjectiveBoundStop`*`)` (operation)

Returns: true

Optimisation will terminate if a feasible solution is found with objective value at least as good as BestObjectiveBoundStop. Note that BestObjectiveBoundStop should be a float, however an integer value can be given which will be automatically converted to a float.

3.7.3 GurobiSetCutOff (for IsGurobiModel, IsFloat)

▷ `GurobiSetCutOff(`*`Model, CutOff`*`)` (operation)

Returns: true

Optimisation will terminate if the objective value is worse than CutOff. Note that CutOff should be a float, an integer value can be given which will be automatically converted to a float.

3.7.4 GurobiSetNumericFocus (for IsGurobiModel, IsInt)

▷ `GurobiSetNumericFocus(`*`Model, NumericFocus`*`)` (operation)

Returns: true

Set the numeric focus for a model. Numeric focus must be in the set [0,1,2,3]. A numeric focus of 0 sets the numeric focus automatically, preferancing speed. Values between 1 and 3 increase the care taken in computations as the value increases, but also take longer. The default value is 0.

3.7.5 GurobiSetMIPFocus (for IsGurobiModel, IsInt)

▷ `GurobiSetMIPFocus(`*`Model, MIPFocus`*`)` (operation)

Returns: true

Set a the MIP focus for a model. The mip focus must be in the set [0,1,2,3], and the default value is 0. The MIP focus alows you to prioritise finding solutions or proving their optimality. See the Gurobi documentation for more information.

3.7.6 GurobiSetBestBoundStop (for IsGurobiModel, IsFloat)

▷ `GurobiSetBestBoundStop(`*`Model, BestBdStop`*`)` (operation)

Returns: true

Set the best bound stopping value for a model. Terminates opmitzation as soon as the value of the best bound is determined to be at least as good as the best bound stopping value.

3.7.7 GurobiSetSolutionLimit (for IsGurobiModel, IsInt)

▷ `GurobiSetSolutionLimit(`*`Model, BestBdStop`*`)` (operation)

Returns: true

Set the limit for the maximum number of MIP solutions to find.

3.7.8 GurobiSetIterationLimit (for IsGurobiModel, IsFloat)

▷ `GurobiSetIterationLimit(`*`Model, IterationLimit`*`)` (operation)

Returns: true

Set the limit for the maximum number of simplex iterations performed.

3.7.9 GurobiSetNodeLimit (for IsGurobiModel, IsFloat)

▷ `GurobiSetNodeLimit(`*`Model, NodeLimit`*`)` (operation)

Returns: true

Set the limit for the maximum number of MIP nodes explored.

3.8 Modifying other attributes and parameters

3.8.1 GurobiSetIntegerParameter

▷ `GurobiSetIntegerParameter(`*`Model, ParameterName, ParameterValue`*`)` (function)

Returns:

Takes a Gurobi model and assigns a value to a given integer-valued parameter. ParameterValue must be a integer value. Refer to the Gurobi documentation for a list of parameters and their types.

3.8.2 GurobiSetDoubleParameter

▷ `GurobiSetDoubleParameter(`*`Model, ParameterName, ParameterValue`*`)` (function)

Returns:

Takes a Gurobi model and assigns a value to a given double-valued parameter. ParameterValue must be a double value. Refer to the Gurobi documentation for a list of parameters and their types.

3.8.3 GurobiSetIntegerAttribute

▷ `GurobiSetIntegerAttribute(`*`Model, AttributeName, AttributeValue`*`)` (function)

Returns:

Takes a Gurobi model and assigns a value to a given integer-valued attribute. AttributeValue must be a double value Refer to the Gurobi documentation for a list of attributes and their types.

3.8.4 GurobiSetDoubleAttribute

▷ `GurobiSetDoubleAttribute(`*`Model, AttributeName, AttributeValue`*`)` (function)

Returns:

Takes a Gurobi model and assigns a value to a given double-valued attribute. AttributeValue must be a double value Refer to the Gurobi documentation for a list of attributes and their types.

3.8.5 GurobiSetDoubleAttributeArray

▷ `GurobiSetDoubleAttributeArray(`*`Model, AttributeName, AttributeValueArray`*`)` (function)

Returns:

Takes a Gurobi model and assigns a value to a given attribute which takes an array of floats. AttributeValue must be an array of floats. Refer to the Gurobi documentation for a list of attributes and their types.

3.9 Other

3.9.1 GurobiWriteToFile

▷ `GurobiWriteToFile(`*`Model, FileName`*`)` (function)

Returns:

Takes a model and writes it to a file. File type written is determined by the FileName suffix. File types include .mps, .rew, .lp, .rlp, .ilp, .sol, or .prm Refer to the gurobi documentation for more infomation on which file types can be read.

3.9.2 GurobiUpdateModel

▷ `GurobiUpdateModel(`*`Model`*`)` (function)

Returns:

Takes a model and updates it. Changes to parameters or constraints are not processed until the model is either updated or optimised.

Chapter 4

Examples

This section contains a number of examples which are intended to illustrate the usage of Gurobify.

4.1 Sudoku solver

The purpose of the example Statement of the problem The method applied Defining the variables with a view to the constraints (all diff) In this instance we name the variables since that is of use to us

Example
```
gap> var_names :=[];
gap> for i in [1 .. 9] do
>      for j in [1 .. 9] do
>        for k in [1 .. 9] do
>          name := Concatenation("x", String(i), String(j), String(k));;
gap>          Add(var_names, name);
gap>        od;
gap>      od;
gap>    od;
```

Create the model. We need to tell Gurobi that each variable is binary.

Example
```
gap> var_types := ListWithIdenticalEntries( Size( var_names ), "Binary" );;
gap> model := GurobiNewModel( var_types, var_names );
<object>
```

Here we define a few basic functions which are purely for the purpose of this example. Firstly a way to go from a subset of the variables to the corresponding index set. Secondly, a way of going back from the index set to identify the variable name. Lastly, a method of displaying the Sudoku board given the variables which are in the solution set.

Example
```
gap> ExampleFuncNamesToIndex := function( vari_names, var_included )
>      local vars, ind;
gap>      vars := ListWithIdenticalEntries( Size( var_names ), 0 );
gap>      ind := List( var_included, t -> Position( var_names, t ));
gap>      vars{ ind }:=ListWithIdenticalEntries( Size( var_included ), 1 );
gap>      return vars;
gap> end;
gap> ExampleFuncIndexToNames := function( var_names, index_set )
```

```
>   return Filtered( var_names, t -> index_set[Position( var_names, t )] = 1. );
gap> end;
gap> ExampleFuncDisplaySudoku := function( sol2 )
>     local mat_sol, m, i, j, k;
gap>        mat_sol := NullMat(9, 0);;
gap>        for m in sol2 do
>          i := EvalString( [m[2]] );
gap>          j := EvalString( [m[3]] );
gap>          k := EvalString( [m[4]] );
gap>          mat_sol[i][j] := k;
gap>        od;
gap>      Display(mat_sol);
gap> end;
```

Ensure that a square only takes a single value.

```
gap> for i in [1 .. 9] do
>     for j in [1 .. 9] do
>       uniqueness_constr :=[];
gap>          for k in [1 .. 9] do
>           name := Concatenation("x", String(i), String(j), String(k));;
gap>            Add(uniqueness_const, name);
gap>          od;
gap>          constr := ExampleFuncNamesToIndex( var_names, uniqueness_constr );
gap>          GurobiAddConstraint( model, constr , "=", 1 );
gap>      od;
gap> od;
```

Ensure that each value occurs exactly once per row.

```
gap> for i in [1 .. 9] do
>     for k in [1 .. 9] do
>       row_constr :=[];
gap>        for j in [1 .. 9] do
>         name := Concatenation("x", String(i), String(j), String(k));;
gap>          Add(row_constr, name );
gap>        od;
gap>        constr := ExampleFuncNamesToIndex( var_names, row_constr );
gap>        GurobiAddConstraint( model, constr, "=", 1 );
gap>      od;
gap> od;
```

Ensure that each value occurs exactly once per column.

```
gap> for j in [1 .. 9] do
>     for k in [1 .. 9] do
>       column_constr :=[];
gap>        for i in [1 .. 9] do
>         name := Concatenation("x", String(i), String(j), String(k));;
gap>          Add(column_constr, name);
gap>        od;
gap>        constr := ExampleFuncNamesToIndex( var_names, column_constr );
gap>        GurobiAddConstraint(model, constr, "=", 1);
gap>      od;
gap> od;
```

Ensure that each value occurs exactly once per sub-square. We start at the top left corner of each square and work our way through them.

Example

```
gap> starter_points := [ [1,1], [1,4], [1,7], [4,1], [4,4],
> [4,7], [7,1], [7,4], [7,7]];
gap> for m in starter_points do
>     for k in [1 .. 9] do
>       square_constr := [];
gap>        for i in [0 .. 2] do
>         for j in [0 .. 2] do
>           name := Concatenation(
>                     "x", String(m[1] + i), String(m[2] + j), String(k)
>                   );;
gap>            Add(square_constr, name);
gap>          od;
gap>        od;
gap>        constr := ExampleFuncNamesToIndex( var_names, square_constr );
gap>        GurobiAddConstraint(model, constr, "=", 1);
gap>      od;
gap> od;
```

Now that the lp file will look for solutions that obey the Sudoku rules, we can put in the inital Sudoku configuration by assigning certain entries of the sudoku matrix set values.

Example

```
gap> starter_squares := ["x112", "x123", "x164", "x217", "x248", "x326",
> "x343", "x391", "x411", "x422", "x488", "x516", "x549", "x568",
> "x595", "x625", "x682", "x693", "x719", "x767", "x785", "x865",
> "x894", "x942", "x986", "x998"];
gap> constr := ExampleFuncNamesToIndex(var_names, starter_squares);;
gap> GurobiAddConstraint( model, constr , "=", Sum( constr ), "StarterSquares");
```

Now we optimize. Change the solution into the variable names, and then display.

Example

```
gap> GurobiOptimizeModel( model );
gap> sol := GurobiSolution( model );;
gap> sol2 := ExampleFuncIndexToNames( var_names, sol );;
gap> ExampleFuncDisplaySudoku( sol2 );
[ [  2,  3,  1,  6,  5,  4,  8,  9,  7 ],
  [  7,  9,  4,  8,  1,  2,  5,  3,  6 ],
  [  5,  6,  8,  3,  7,  9,  2,  4,  1 ],
  [  1,  2,  7,  5,  3,  6,  4,  8,  9 ],
  [  6,  4,  3,  9,  2,  8,  7,  1,  5 ],
  [  8,  5,  9,  7,  4,  1,  6,  2,  3 ],
  [  9,  1,  6,  4,  8,  7,  3,  5,  2 ],
  [  3,  8,  2,  1,  6,  5,  9,  7,  4 ],
  [  4,  7,  5,  2,  9,  3,  1,  6,  8 ] ]
```

At this point we may wish to save the model as an lp file so that other Sudoku problems may be quickly and easily solved in the future. Of course, we do not want to save the starter configuration, only the general Sudoku constraints, and so we must first delete the the constraint "StarterSquares".

Example

```
gap> GurobiDeleteConstraintsWithName( model, "StarterSquares" );
gap> GurobiWriteToFile( model, "SudokuSolver.lp" );
```

We can now load the lp file to create a new model with all the generic Sudoku constraints. Assuming we have defined the functions ExampleFuncNamesToIndex, ExampleFuncIndexToNames and ExampleFuncDisplaySudoku as before, we may simply add a new constraint to the model to represent the starting configuration of the Sudoku problem. Assuming we do not remember the variable names or their order, we must first extract this information from the model. We then optimize the model and display the solution as before. Incase we have forgotten the names of the variables, or the order they occur, or simply don't want to reconstruct the var_names list, we can first extract this information directly from the model.

Example

```
gap> model2 := GurobiReadModel( "SudokuSolver.lp" );
gap> var_names2 := GurobiVariableNames(model2);
gap> starter_squares := ["x118", "x124", "x132", "x145", "x161", "x219",
>  "x337", "x353", "x414", "x425", "x441", "x539", "x544", "x562",
>  "x573", "x669", "x686", "x691", "x754", "x778", "x894", "x947",
>  "x968", "x971", "x985", "x999"];
gap> constr := ExampleFuncNamesToIndex(var_names2, starter_squares);;
gap> GurobiAddConstraint(model2, constr , "=", Sum( constr ));
gap> GurobiOptimizeModel(model2);
gap> sol := GurobiSolution(model2);;
gap> sol2 := ExampleFuncIndexToNames(var_names2, sol);;
gap> ExampleFuncDisplaySudoku( sol2 );
[ [  8,  4,  2,  5,  9,  1,  7,  3,  6 ],
  [  9,  3,  1,  6,  2,  7,  5,  4,  8 ],
  [  5,  6,  7,  8,  3,  4,  9,  1,  2 ],
  [  4,  5,  3,  1,  8,  6,  2,  9,  7 ],
  [  6,  1,  9,  4,  7,  2,  3,  8,  5 ],
  [  2,  7,  8,  3,  5,  9,  4,  6,  1 ],
  [  1,  9,  6,  2,  4,  5,  8,  7,  3 ],
  [  7,  8,  5,  9,  1,  3,  6,  2,  4 ],
  [  3,  2,  4,  7,  6,  8,  1,  5,  9 ] ]
```

What if we removed a initial value from the Sudoku problem? How many solutions would there be? We remove set an entry from the starter configuration and then optimize. We feed this solution back in as a constraint, and then reoptimize. We can repeat this process until we have found all feasible solutions, which will be when the model becomes infeasible.

Example

```
gap> model3 := GurobiReadModel( "SudokuSolver.lp" );
gap> var_names3 := GurobiVariableNames(model3);
gap> starter_squares := ["x118", "x124", "x132", "x145", "x161", "x219",
>  "x337", "x353", "x414", "x425", "x441", "x539", "x544", "x562",
>  "x573", "x669", "x686", "x691", "x754", "x778", "x894", "x947",
>  "x968", "x971", "x985"];
gap> constr := ExampleFuncNamesToIndex(var_names3, starter_squares);;
gap> GurobiAddConstraint( model3, constr , "=", Sum( constr ));
gap> GurobiOptimizeModel( model3 );
gap> if GurobiStatus( model3 ) = 2 then
>         number_of_solutions := 1;
gap> else
>         number_of_solutions := 0;
gap> fi;
gap> while GurobiStatus( model3 ) = 2 do
>         sol := GurobiSolution( model3 );;
gap>         number_of_solutions := number_of_solutions + 1;
gap>         GurobiAddConstraint( model3, sol , "<", 80 );
gap>         GurobiOptimizeModel( model3 );
gap> od;
gap> Print( number_of_solutions, "\n");
67
```

Chapter 5

Appendix

5.1 Links to some Gurobi documentation

For more information on Gurobi parameters, attributes, and status codes, see the following links:

- Attributes: http://www.gurobi.com/documentation/7.0/refman/attributes.html
- Parameters: http://www.gurobi.com/documentation/7.0/refman/parameters.html
- Status codes: https://www.gurobi.com/documentation/7.0/refman/optimization_status_codes.html
- Licencing: http://www.gurobi.com/products/licensing-pricing/licensing-overview

5.2 Optimisation Status Codes

The information in this section on optimisation status codes is taken directly from the Gurobi website[gur16d].

- 1 - Model is loaded, but no solution information is available.
- 2 - Model was solved to optimality (subject to tolerances), and an optimal solution is available.
- 3 - Model was proven to be infeasible.
- 4 - Model was proven to be either infeasible or unbounded. To obtain a more definitive conclusion, set the DualReductions parameter to 0 and reoptimize.
- 5 - Model was proven to be unbounded. Important note: an unbounded status indicates the presence of an unbounded ray that allows the objective to improve without limit. It says nothing about whether the model has a feasible solution. If you require information on feasibility, you should set the objective to zero and reoptimize.
- 6 - Optimal objective for model was proven to be worse than the value specified in the Cutoff parameter. No solution information is available.
- 7 - Optimization terminated because the total number of simplex iterations performed exceeded the value specified in the IterationLimit parameter, or because the total number of barrier iterations exceeded the value specified in the BarIterLimit parameter.

- 8 - Optimization terminated because the total number of branch-and-cut nodes explored exceeded the value specified in the NodeLimit parameter.
- 9 - Optimization terminated because the time expended exceeded the value specified in the TimeLimit parameter.
- 10 - Optimization terminated because the number of solutions found reached the value specified in the SolutionLimit parameter.
- 11 - Optimization was terminated by the user.
- 12 - Optimization was terminated due to unrecoverable numerical difficulties.
- 13 - Unable to satisfy optimality tolerances; a sub-optimal solution is available.
- 14 - An asynchronous optimization call was made, but the associated optimization run is not yet complete.
- 15 - User specified an objective limit (a bound on either the best objective or the best bound), and that limit has been reached.

References

[GAP16] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.8.6*, 2016. 4

[GH16] Sebastian Gutsche and Max Horn. *AutoDoc – a GAP package, Version 2016.03.08*, 2016. 2

[gur16a] Gurobi Optimization, Inc. *Gurobi Optimizer Reference Manual*, 2016. 4

[gur16b] Gurobi Optimizer Academic Program. http://www.gurobi.com/pdfs/Pricing-Academic.pdf, 2016. [Online; accessed 17-March-2017]. 5

[gur16c] Licensing Overview. http://www.gurobi.com/products/licensing-pricing/licensing-overview, 2016. [Online; accessed 17-March-2017]. 5

[gur16d] Optimization Status Codes. https://www.gurobi.com/documentation/7.0/refman/optimization_status_codes.html, 2016. [Online; accessed 21-March-2017]. 23

[Hor16] Max Horn. *PackageMaker – a GAP package, Version 0.8*, 2016. 2

Index