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 Welcome to Gurobify

Gurobify is a GAP[GAP16] package which provides an interface to the optimisation software Gurobi[gur16a].

INSERT LICENCE INFO

Gurobify can be obtained from the Gurobify homepage:

- https://github.com/jesselansdown/Gurobify

Please use the issue tracker to inform me of any bugs or suggestions:

- https://github.com/jesselansdown/Gurobify/issues

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 Testing and documentation

To test the package is working correctly, run the following command from inside the Gurobify directory.

Example

```
gap.sh maketest.g
```

It transforms each of the examples in Chapter 4 "Examples" and the example given in Section 2.2 "Simple example" into test cases to check that Gurobify is functioning correctly.

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 format. 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
```

Regenerating the documentation will also automatically regenerate the "maketest.g" test file.

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

This section contains a simple example which demonstrates some of the key functionality of Gurobify. We demonstrate the following:

- Creation of a Gurobi model, including specifying variable types and names
- Adding an objective function to the model and choosing its sense (maximise or minimise)
- Adding constraints to a model individually
- Adding multiple constraints to a model at one time
- Deleting constraints from a model
- Optimizing a model
- Querying the optimisation status of a model
- Displaying the solution to a feasible model
- Setting a parameter (in this case the time limit)
- Querying the current value of a parameter (again, the time limit)
- Writing a model to a file
- Reading a model from a file

The purpose of this example is to illustrate the use of Gurobify in a quick, simple manner. The example itself is trivial, but the same proceedure applies for much more complex models. It only utilises some of the core commands available in Gurobify, though more commands are listed in detail in Chapter 3 "Using Gurobify", and further examples are given in Chapter 4 "Examples".

2.2 Simple example

The first step in using gurobify is to create a model. To do this, we first need to define the types of variables that are to be used in the model, and we may optionally give them names. In this case we create a model with three binary variables, called x, y and z. We then define the objective function as $x+2y+z$, which we set to be maximised.

Example
```
gap> model := GurobiNewModel(["BINARY", "BINARY", "BINARY"], ["x", "y", "z"]);
<object>
gap> GurobiSetObjectiveFunction( model, [1.,2.,1.] );
true
gap> GurobiMaximiseModel(model);
true
```

Having defined our model, we can now add constraints. To do this, a list of the coefficients for each of the variables is given, along with the sense of the equation (that is, $=$, $\leq$ or $\geq$), and the value on the right hand side of the constraint, and optionally a name for the constraints. Gurobi does not distinguish between $\leq$ and $<$ or $\geq$ and $>$, though we note that gurobify only accepts the form $<$ or $>$. We add constraints the $2x+2y+2z \leq 6$ and $x+2y+3z \geq 5$ to our model. We do not assign them names.

Example
```
gap> GurobiAddConstraint(model, [2, 2, 2], "<", 6 );
true
gap> GurobiAddConstraint(model, [1, 2, 3], ">", 5 );
true
```

Alternatively, the previous constraints could have been added simultaneously, by containing multiple constraints as entries of a list.

Example
```
gap> constraints :=  [[2, 2, 2], [1, 2, 3]];
[ [ 2, 2, 2 ], [ 1, 2, 3 ] ]
gap> sense := ["<", ">"];
[ "<", ">" ]
gap> rhs := [6,5];
[ 6, 5 ]
gap> GurobiAddMultipleConstraints( model, constraints, sense, rhs );
true
```

We can now optimize the model. This will return an Gurobi opimiztion status code. More information on the status codes can be found in the Appendix. A status code of 2 lets us know that the model was successfully optimized. We may then query the model's solution.

Example
```
gap> GurobiOptimizeModel(model);
2
gap> GurobiSolution(model);
[ 1., 1., 1. ]
```

In addition to returning the optimisation status upon finishing optimisation, we can query the optimisation status of a model directly at any point in time. It will give the status of the model at the point when it was last optimised.

Example
```
gap> GurobiOptimizationStatus(model);
2
```

We can reset any information found on a model to its pre-optimization stat. If we then check its status, it will tell us that the model has been loaded, but no optimization information is available.

Example
```
gap> GurobiReset(model);
true
gap> GurobiOptimizationStatus(model);
1
```

We can change the objective sense of the model so that Gurobi will look for a solution which minimises the objective function instead. We then optimise the model and, if the optimisation is succesful, retun the solution.

Example
```
gap> GurobiMinimiseModel(model);
true
gap> GurobiOptimizeModel(model);
2
gap> GurobiSolution(model);
[ 0., 1., 1. ]
```

We can write the model to a file so that we may use it later. We need to specify the file name, and the extension of the file name will determine the type of file written. In this case we write the model to an lp file which we call "test.lp".

Example
```
gap> GurobiWriteToFile(model, "test.lp");
true
```

It is also possible to read a model directly from a file. In this case it is not necessary to build the model from the ground up. We will read in the model from the "test.lp" file that we created in the previous step.

Example
```
gap> re_model := GurobiReadModel("test.lp");
<object>
```

We now add another constraint, $x+y+z>3$, to the model. Since we may want to remove this constraint later we give it the name "Other Constraint". We optimise the model and since it returns a feasible optimisation status we return the solution.

Example
```
gap> GurobiAddConstraint(re_model, [1, 1, 1], ">", 3, "Other Constraint");
true
gap> GurobiOptimizeModel(re_model);
2
gap> GurobiSolution(re_model);
[ 1., 1., 1. ]
```

We add another constraint, $y+z<1$, to the model and also call it "Other Constraint". We optimise the model, and get a status code of 3, indicating that the model has no feasible solutions.

Example
```
gap> GurobiAddConstraint(re_model, [0, 1, 1], "<", 1, "Other Constraint");
true
gap> GurobiOptimizeModel(re_model);
3
```

Since we named the two additional constraints, we can delete them. This makes our model feasible again, and we get the same solution as before.

Example
```
gap> GurobiDeleteConstraintsWithName(re_model, "Other Constraint");
true
gap> GurobiOptimizeModel(re_model);
2
gap> GurobiSolution(re_model);
[ 0., 1., 1. ]
```

There are many parameters and attributes of Gurobi models which can be set and queried. For example, we may set the time limit to something very small so that Gurobi terminates before finishing optimising. This returns a status code of 9. We may then change the time limit again to allow Gurobi more time to finish optimising and thus obtain a feasible solution again. Chapter 3 documents other parameters and attributes that Gurobify is able to modify.

Example
```
gap> GurobiSetTimeLimit(re_model, 0.000001);
true
gap> GurobiOptimizeModel(re_model);
9
gap> GurobiSetTimeLimit(re_model, 10);
true
gap> GurobiOptimizeModel(re_model);
2
```

We can query the model to find out what the current time limit value is set to.

Example
```
gap> GurobiTimeLimit(re_model);
10.
```

Chapter 3

Using Gurobify

This chapter documents the various commands available in Gurobify. Each command begins with "Gurobi", which in some instances helps to avoid conflict with other GAP commands, and is used for consistancy in other instances. Note that a Gurobi model object in GAP is simply a pointer to the Gurobi model within Gurobi itself. Thus care must be taken not to overwrite the Gurobi model without first releasing the memory associated with it by calling the TODO command. It is also important to note that many changes to a model do not become active until the model is updated. This can be done by either Optimizing the model, or calling GurobiUpdateModel. It is often unnecessary to update the model if it will later be optimized, and in fact can be more efficient to only update when necessary. However, one may occasionally wish to perform a command which is dependent on previous changes to the model. In this case it is necessary to call the update command first.

3.1 Creating or reading a model

This section deals with reading, writing, and creating models, as well as working with the model variables.

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 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.

3.1.3 GurobiNewModel (for IsList, IsList)

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

Returns: A Gurobi model

Creates a gurobi model with variables defined by VariableTypes. VariableTypes must be a list of strings, where each entry is the type of the corresponding variable. Accepted variable types are "CONTINUOUS", "BINARY", "INTEGER", "SEMICONT", or "SEMIINT". Refer to the Gurobi documentation for more information on the variable types. Optionally takes the names of the variables as a list of strings.

3.1.4 GurobiSetVariableNames (for IsGurobiModel, IsList)

▷ `GurobiSetVariableNames(Model, VariableNames)` (operation)

Returns: true

To do: check that everything is a string

3.2 Adding and deleting constraints

This section deals with adding linear constraints, both individually and in bulk, naming constraints, and also deleting constraints.

3.2.1 GurobiDeleteAllConstraintsWithName

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

Returns: true

Deletes all constraints from a model with the name ConstraintName.

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

▷ `GurobiAddConstraint(Model, ConstraintEquation, ConstraintSense, ConstraintRHSValue[, ConstraintName])` (operation)

Returns: true

Same as below, except that ConstraintRHS value takes an integer value.

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

▷ `GurobiAddConstraint(Model, ConstraintEquation, ConstraintSense, ConstraintRHSValue[, ConstraintName])` (operation)

Returns: true

Adds a constraint to a gurobi model. ConstraintEquation must be a list, such that each entry is the coefficient (including 0 coefficents) of the corresponding variable in the constraint equation. The ConstraintSense must be one of "<", ">" or "=", where Gurobi interprets < as <= and > as >=. The ConstraintRHSValue is the value on the right hand side of the constraint. A constraint may optionally 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.

3.2.4 GurobiAddMultipleConstraints (for IsGurobiModel, IsList, IsList, IsList, IsList)

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

Returns: true

Add multiple constraints to a model at one time. The arguments (except Model) are lists, such that the i-th entries of each list determine a single constraint in the same manner as for the operation GurobiAddConstraint.

3.3 Adding and modifying objective functions

This section deals with adding and modifying objective functions to a model, and changing between maximising and minimising 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. ObjectiveValues is a list of coefficients (including 0 coefficeints) corresponding to each of the variables

3.3.4 GurobiObjectiveFunction (for IsGurobiModel)

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

Returns: List of coefficients of the objective function

View the objectivive function for a model.

3.4 Optimizing a model

This section deals with optimizing a model, and handling solution information.

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, or alternatively see the Appendix of this manual. The model itself is altered to reflect the optimisation, and more information about it can be obatained using other functions, in particular the GurobiSolution and GurobiObjectiveValue functions.

3.4.2 GurobiReset

▷ `GurobiReset(Model)` (function)

Returns:

Reset all information associated with a solution for the model.

3.4.3 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.

3.4.4 GurobiSolution (for IsGurobiModel)

▷ `GurobiSolution(Model)` (operation)

Returns: Solution

Display the solution found for a successfuly optimised model.

3.4.5 GurobiObjectiveValue (for IsGurobiModel)

▷ `GurobiObjectiveValue(Model)` (operation)

Returns: objective value

Returns the objective value of the current solution.

3.4.6 GurobiOptimizationStatus (for IsGurobiModel)

▷ `GurobiOptimizationStatus(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, or alternatively refer to the Appendix of this manual.

3.5 Querying attributes and parameters

This section deals with obtaining information about attributes and parameters of a Gurobi model. 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. Note also that the attributes are Gurobi attributes, and are not true attributes in the GAP sense. Crucially, attributes for a model constantly change, either as a result of optimization or from manually setting them. Thus attributes for Gurobi models are approximated by Gurobify using GAP operations or functions. Their usage should still be comfortable for users familiar with GAP attributes.

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 GurobiObjectiveBound (for IsGurobiModel)

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

Returns: objective bound

Returns the best known bound for the model.

3.5.4 GurobiRunTime (for IsGurobiModel)

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

Returns: run time of optimization

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

3.5.5 GurobiNumericFocus (for IsGurobiModel)

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

Returns: numeric focus

Returns the numeric focus value of the model.

3.5.6 GurobiTimeLimit (for IsGurobiModel)

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

Returns: time limit

Returns the time limit for the model.

3.5.7 GurobiCutOff (for IsGurobiModel)

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

Returns: cutoff value

Returns the cutoff value for the model.

3.5.8 GurobiBestObjectiveBoundStop (for IsGurobiModel)

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

Returns: best objective bound limit value

Returns the best objective bound limit value for the model.

3.5.9 GurobiMIPFocus (for IsGurobiModel)

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

Returns: MIP focus

Returns the MIP focus value for the model.

3.5.10 GurobiBestBoundStop (for IsGurobiModel)

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

Returns: Best bound stopping value

Returns the best bound stopping value for the model.

3.5.11 GurobiSolutionLimit (for IsGurobiModel)

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

Returns: solution limit value

Returns the solution limit value for the model.

3.5.12 GurobiIterationLimit (for IsGurobiModel)

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

Returns: Iteration limit

Returns the iteration limit value for the model.

3.5.13 GurobiNodeLimit (for IsGurobiModel)

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

Returns: Node limit

Returns the node limit value for the model.

3.5.14 GurobiVariableNames (for IsGurobiModel)

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

Returns:

Returns the names of the variables in the model. The order of the variables acts as the index list for any lists of variable coefficients, such as in GurobiAddConstraint or GurobiSetObjectiveFunction.

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 specific value types can be queried using the parameter or attribute name as a strings given exactly as in the Gurobi documentation. See the Appendix for links to the relevant documentation.

3.6.1 GurobiIntegerParameter

▷ `GurobiIntegerParameter(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 GurobiDoubleParameter

▷ `GurobiDoubleParameter(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 GurobiIntegerAttributeArray

▷ `GurobiIntegerAttributeArray(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 values, Refer to the Gurobi documentation for a list of attributes and their types.

3.6.6 GurobiDoubleAttributeArray

▷ `GurobiDoubleAttributeArray(Model, AttributeName)` (function)

Returns: attibute array

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

3.6.7 GurobiStringAttributeArray

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

Returns: attibute array

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

3.7 Modifying attributes and parameters

This section deals with modifying the values of attributes and parameters of Gurobi models. Note that a model must be updated or optimized before parameters and attributes are updated, any queries or commands which depend on these values will use the values at the time the model was last updated or optimized, even if the values have since been modified.

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

In addition to these specific commands given in the previous section, other gurobi parameters and attributes which take specific value types can be modified using the parameter or attribute name as a strings given exactly as in the Gurobi documentation. See the Appendix for links to the relevant documentation.

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.

Chapter 4

Examples

This section contains a number of examples which are intended to illustrate the usage of Gurobify. Each of these examples also form a test suite (along with the Simple example in Chapter 2), which can be used to check that Gurobify is functioning properly. See Section 1.5 "Testing and documentation" for more on this aspect of the examples.

4.1 Sudoku solver

To solve a sudoku puzzle, the integers from 1 to 9 must be placed in each cell of a 9×9 grid, such that every number in a column, row, or one of the nine subgrids, is different. A starting configuration is given which must be incorporated into the final solution. In this example we create a model which imposes the sudoku rules as constraints, takes an additional constraint for the starting configuration of the Sudoku puzzle, and then solves the puzzle.

To begin with we will define the variables that we will need for our model. Each variable will have a name of the form x_{ijk}, where i is the row of the sudoku puzzle, j is the column, and k is the value of the entry defined byt cell ij. The variables are binary, since a value of 1 indicates that the corresponding cell ij has value k, and a 0 indicates that it doesn't have that value.

Since the variable names are important to the fomulation of this problem, we must define the variable names.

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;
```

Now 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 the names of 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 from the names of 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.

Example
```
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_constr, 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.

Example
```
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.

Example
```
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;
```

The model now has constraints that will ensure that the Sudoku rules are obeyed. We can put in the inital Sudoku configuration by assigning values to certain entries of the sudoku matrix.

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");
true;
```

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

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

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. In case we do not remember the variable names or their order, we can first extract this information from the model. We then optimize the model and display the solution as before.

Example
```
gap> model2 := GurobiReadModel( "SudokuSolver.lp" );
<object>
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 ));
true
gap> GurobiOptimizeModel(model2);
2
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" );
<object>
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 ));
true
gap> GurobiOptimizeModel( model3 );
2
gap> if GurobiOptimizationStatus( model3 ) = 2 then
>         number_of_solutions := 1;
gap> else
>         number_of_solutions := 0;
gap> fi;
gap> while GurobiOptimizationStatus( 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 Gurobify links

- Homepage: https://github.com/jesselansdown/Gurobify
- Issue tracker: https://github.com/jesselansdown/Gurobify/issues

5.2 GAP links

- Homepage: http://gap-system.org/

5.3 Gurobi links

- Homepage: http://gurobi.com/

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

5.4 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]. 27

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

Index