Gurobify

A GAP interface to Gurobi Optimizer.

2.0.0

24 June 2021

Jesse Lansdown

Jesse Lansdown
Email: jesse.lansdown@uwa.edu.au
Homepage: https://www.jesselansdown.com
Address: Jesse Lansdown
Centre for the Mathematics of Symmetry and Computation
The University of Western Australia
35 Stirling Highway
PERTH WA 6009
Australia

Abstract

Gurobify provides an interface to the Gurobi Optimizer software from GAP. It enables the creation and modification of mixed integer and linear programming 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[GH17] 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].

Gurobify can be obtained from the Gurobify homepage:

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

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

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

1.2 Citing Gurobify

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. Check that the version number and doi correspond to the version that you used in your research. Please also send me an email informing me of the paper for my own interest!

Example

```
@article{gurobify,
Author = {Lansdown, Jesse},
Doi = {10.5281/zenodo.5025558},
Key = {gurobify},
Title = {{Gurobify -- A GAP interface to Gurobi Optimizer, Version 2.0.0}},
Url = {http://doi.org/10.5281/zenodo.5025558},
Year = 2021,
}
```

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

Example

```
\bibitem[Lansdown(2021)]{gurobify}
J.~Lansdown.
\newblock {Gurobify -- A GAP interface to Gurobi Optimizer, Version 2.0.0}.
```

```
\newblock 2021.
\newblock \doi{10.5281/zenodo.5025558}.
\newblock URL \url{http://doi.org/10.5281/zenodo.5025558}.
```

1.3 Dependencies

Gurobify requires the following software to be installed.

- GAP 4.8 (or later)
- Gurobi 7.0 (or later)
- 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 2020.08.11 (or later)
- GAPDoc 1.6.4 (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 extract 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 privileges are needed to access the GAP root directory. Next run the following command in the terminal from within the Gurobify directory.

Example

```
./install.sh -g <gurobi path> [-r <gap path>]
```

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

-g /Library/gurobi701/mac64/

or something like this on Linux,

-g /opt/gurobi701/linux64/

The [-r <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,

-r /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 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 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 the following.

Example

```
gap>  LoadPackage("Gurobify");
------------------------------------------------------------------------------
Loading  Gurobify 2.0.0 (Gurobify provides an interface to Gurobi from GAP.)
by Jesse Lansdown (www.jesselansdown.com).
Homepage: https://www.jesselansdown.com/Gurobify/
     Currently running Gurobi Optimizer 9.1.0
------------------------------------------------------------------------------
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 -l ";/alternative/path/to/Gurobify"
```

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

1.7 Problems With The Installation

Please ensure that all dependencies are present, the correct paths have been used, and the correct versions of software are being used. There is also a section in the Appendix, "Troubleshooting The Installation", which deals with a number of issues people face when installing Gurobify, and suggests solutions.

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
- Optimising 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 procedure 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"]);
Gurobi model
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, $=$, $<$ or $>$), the value on the right hand side of the constraint, and optionally a name for the constraints. Gurobi does interprets $<$ as $\leq$ and $>$ as $\geq$ and does not distinguish between them. Note, however, that gurobify only accepts the form $<$ or $>$. We add the constraints $2x+2y+2z \leq 6$ 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 optimise the model. This will return a Gurobi opimistion 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 optimised. We may then query the model's solution.

Example

```
gap> GurobiOptimiseModel(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> GurobiOptimisationStatus(model);
2
```

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

Example

```
gap> GurobiReset(model);
true
gap> GurobiOptimisationStatus(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> GurobiOptimiseModel(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");
Gurobi model
```

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> GurobiOptimiseModel(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> GurobiOptimiseModel(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> GurobiOptimiseModel(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> GurobiOptimiseModel(re_model);
9
gap> GurobiSetTimeLimit(re_model, 10);
true
gap> GurobiOptimiseModel(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. It is important to note that many changes to a model do not become active until the model is updated. This can be done by either optimising the model, or calling GurobiUpdateModel. It is often unnecessary to update the model if it will later be optimised, 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: true

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 written.

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". The variable types are not case sensitive. 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 GurobiNewModel (for IsPosInt, IsString)

▷ `GurobiNewModel(`*`n, VariableType`*`)` (operation)

Returns: A Gurobi model

Creates a new Gurobi model, with n variables, all of type VariableType. Accepted variable types are "CONTINUOUS", "BINARY", "INTEGER", "SEMICONT", or "SEMIINT". The variable types are not case sensitive. Refer to the Gurobi documentation for more information on the variable types.

3.1.5 GurobiSetVariableNames (for IsGurobiModel, IsList)

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

Returns: true

Assigns each variable a new name from a list of names. The names must be strings.

3.2 Adding And Deleting Constraints

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

3.2.1 GurobiDeleteSingleConstraintWithName

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

Returns: true

Deletes a single constraint from a model with the name ConstraintName. If multiple constraints have this name, then one will be deleted at random.

3.2.2 GurobiDeleteConstraints

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

Returns: true

Deletes all constraints from a model which are indexed by the values of ConstraintList. Requires an update to the model to take effect.

3.2.3 GurobiDeleteConstraintsWithName (for IsGurobiModel, IsString)

▷ `GurobiDeleteConstraintsWithName(`*`Model, ConstraintName`*`)` (operation)

Returns: true/false

Deletes all constraints with the name ConstraintName

3.2.4 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.

3.2.5 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. If no constraint name is given, then a constraint is simply assigned the name "UnNamedConstraint". Note that a model must be updated or optimised before any additional constraints become effective.

3.2.6 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. ConstraintNames is an optional argument, and must be given for all constraints, or not at all.

3.2.7 GurobiAddMultipleConstraints (for IsGurobiModel, IsList, IsString, IsFloat)

▷ GurobiAddMultipleConstraints(*Model, ConstraintEquations, ConstraintSense, ConstraintRHSValue[, ConstraintName]*) (operation)

Returns: true

Add multiple constraints to a model at one time, where the sense, rhs, and optionally name, of each constraint is the same. ConstraintEquations must be a list of constraints, ConstraintSense must be a string, ConstraintRHSValue must be a float or an integer. ConstraintName is an optional argument, which if given, must be a string.

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 optimised, 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 optimised, 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 objective function for a model.

3.4 Optimising A Model

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

3.4.1 GurobiOptimiseModel

▷ `GurobiOptimiseModel(`*`Model`*`)` (function)

Returns: Optimisation status code.

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 commands, in particular the GurobiSolution and GurobiObjectiveValue functions.

3.4.2 GurobiReset

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

Returns: true

Reset all information associated with an optimisation of the model, such as the optimisation status, the solution and the objective value.

3.4.3 GurobiUpdateModel

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

Returns: true

Takes a model and updates it. Changes to a model (such as 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. Note that if a solution has not been optimised, is infeasible, or the optimisation was not completed, then this will return an error. Thus it is advisable to first check the optimisation status.

3.4.5 GurobiObjectiveValue (for IsGurobiModel)

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

Returns: objective value

Returns the objective value of the current solution.

3.4.6 GurobiOptimisationStatus (for IsGurobiModel)

▷ `GurobiOptimisationStatus(`*`Model`*`)` (operation)

Returns: optimisation status code

Returns the optimisation status code of the most recent optimisation. Refer to the Gurobi documentation for more on the optimisation statuses, or alternatively refer to the Appendix of this manual.

3.4.7 GurobiFindAllBinarySolutions (for IsGurobiModel, IsPosInt)

▷ `GurobiFindAllBinarySolutions(`*`Model, Size`*`)` (operation)

Returns: Set of all solutions.

This function finds all possible solutions of a given size, for a model with only binary variables. Takes a Gurobi model and repeatedly optimises it, each time adding the previous solution as a constraint so that it isn't found again. This continues until all solutions are found, and then they are returned as a set. During the process the number of found solutions is displayed. Note: - Only for models where every variable is a binary variable. - Only finds solution sets of a given size.

3.4.8 GurobiFindAllBinarySolutions (for IsGurobiModel, IsPosInt, IsGroup)

▷ `GurobiFindAllBinarySolutions(`*`Model, Size, Group`*`)` (operation)

Returns: Set of all solutions.

This function finds all possible solutions of a given size, for a model with only binary variables. Same as above, except that it also takes a permutation group acting on the index set of variables. Instead of finding all solutions directly, the group is used to find the orbit of each new solution, and

these are then all returned at the end, and used as constraints until then. An option value may also be given which will only return the representatives of each orbit of the solutions. Hence it returns all the unique solutions up to equivalence under the group. This saves on memory, and the remaing solutions may be refound by generating the orbit under the group. To invoke this option place a colon after the group argument and then put representatives:=true so for example GurobiFindAllSolutions(model, size, gp : representatives:=true);

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 optimised before parameters and attributes are updated - any queries will return the values at the time the model was last updated or optimised. 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 optimisation 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 on the objective value of the model.

3.5.4 GurobiRunTime (for IsGurobiModel)

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

Returns: run time of optimisation

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

3.5.5 GurobiNumericFocus (for IsGurobiModel)

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

Returns: numeric focus

Returns the numeric focus value of the model. The numeric focus is a value in the set [0,1,2,3]. A numeric focus of 0 sets the numeric focus automatically, preferencing 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.

3.5.6 GurobiTimeLimit (for IsGurobiModel)

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

Returns: time limit

Returns the time limit for the model. The default value is infinity.

3.5.7 GurobiCutOff (for IsGurobiModel)

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

Returns: cutoff value

Returns the cutoff value for the model. Optimisation will terminate if the objective value is worse than CutOff. The default value is infinity for minimisation, and negative infinity for maximisation.

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. Optimisation will terminate if a feasible solution is found with objective value at least as good as the best objective bound. The default value is negative infinity.

3.5.9 GurobiMIPFocus (for IsGurobiModel)

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

Returns: MIP focus

Returns the MIP focus value for the 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.

3.5.10 GurobiBestBoundStop (for IsGurobiModel)

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

Returns: Best bound stopping value

Returns the best bound stopping value for the model. Optimisation terminates as soon as the value of the best bound is determined to be at least as good as the best bound stopping value. Default value is infinity.

3.5.11 GurobiSolutionLimit (for IsGurobiModel)

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

Returns: solution limit value

Returns the solution limit value for the model. This value limits the maximum number of MIP solutions that will be found. Default value is $2,000,000,000$.

3.5.12 GurobiIterationLimit (for IsGurobiModel)

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

Returns: Iteration limit

Returns the iteration limit value for the model, which limits the number of simplex iterations performed during optimisation. Default value is infinity.

3.5.13 GurobiNodeLimit (for IsGurobiModel)

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

Returns: Node limit

Returns the node limit value for the model, which limits the number of MIP nodes explored during optimisation. The default value is infinity.

3.5.14 GurobiVariableNames (for IsGurobiModel)

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

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

3.5.15 GurobiLogToConsole (for IsGurobiModel)

▷ `GurobiLogToConsole(`*`Model`*`)` (operation)

Returns: true/false

Returns true if the Gurobi output is set to display to the screen, and false if the output is supressed. Gurobify suppresses the output by default.

3.5.16 GurobiVariableTypes (for IsGurobiModel)

▷ `GurobiVariableTypes(`*`Model`*`)` (operation)

Returns the types of the variables in the model.

3.5.17 GurobiMethod (for IsGurobiModel)

▷ `GurobiMethod(`*`Model`*`)` (operation)

Returns: MethodType

Returns the method used to solve a model. See the Gurobi documentation for more details.

3.5.18 GurobiThreads (for IsGurobiModel)

▷ `GurobiThreads(`*`Model`*`)` (operation)

Returns: ThreadCount

Returns number of threads Gurobi is allowed to use. See the Gurobi documentation for more details.

3.6 Querying Other Attributes And Parameters

In addition to the specific queries given in the previous section, other gurobi parameters and attributes of specific value types can also be queried. The parameter or attribute name must be given as a string exactly as stated 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 retrieves 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 retrieves 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: attribute value

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

3.6.4 GurobiStringAttributeElement

▷ `GurobiStringAttributeElement(`*`Model, position, AttributeName`*`)` (function)

Returns: attribute value

Takes a Gurobi model and retrieves the value of a string attribute element at a given position. For example to get the names of constraints with "ConstrName". Refer to the Gurobi documentation for a list of attributes and their types.

3.6.5 GurobiDoubleAttribute

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

Returns: attribute value

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

3.6.6 GurobiIntegerAttributeArray

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

Returns: attribute array

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

3.6.7 GurobiDoubleAttributeArray

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

Returns: attribute array

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

3.6.8 GurobiStringAttributeArray

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

Returns: attribute array

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

3.6.9 GurobiCharAttributeArray

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

Returns: attribute array

Takes a Gurobi model and retrieves an attribute array. Can only get values of attributes arrays which have char 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 optimised 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 optimised, 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, preferencing 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 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 optimisation as soon as the value of the best bound is determined to be at least as good as the best bound stopping value. Default value is infinity.

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. Default value is $2,000,000,000$.

3.7.8 GurobiSetIterationLimit (for IsGurobiModel, IsFloat)

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

Returns: true

Set the limit for the maximum number of simplex iterations performed. Default value is infinity.

3.7.9 GurobiSetNodeLimit (for IsGurobiModel, IsFloat)

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

Returns: true

Set the limit for the maximum number of MIP nodes explored. The default value is infinity.

3.7.10 GurobiSetLogToConsole (for IsGurobiModel, IsBool)

▷ `GurobiSetLogToConsole(`*`Model, Switch`*`)` (operation)

Returns: true

Turns console logging on or off. If Switch is true the output of Gurobi will be printed to the screen, and if it is false the output will be supressed. The default for Gurobify is to supress the output.

3.7.11 GurobiSetMethod (for IsGurobiModel, IsInt)

▷ `GurobiSetMethod(`*`Model, MethodType`*`)` (operation)

Returns: true

Set the method used to solve a model. -1=automatic (this is the default), 0=primal simplex, 1=dual simplex, 2=barrier, 3=concurrent, 4=deterministic concurrent, 5=deterministic concurrent simplex. See the Gurobi documentation for more details.

3.7.12 GurobiSetThreads (for IsGurobiModel, IsInt)

▷ `GurobiSetThreads(`*`Model, ThreadCount`*`)` (operation)

Returns: true

Set the number of threads Gurobi is allowed to use. The default value is 0, which will use as many cores as it wants. See the Gurobi documentation for more details.

3.8 Modifying Other Attributes And Parameters

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

3.8.1 GurobiSetIntegerParameter

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

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)

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)

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)

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)

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 Additional Functionality

This section provides information on some additional functionality which does not directly relate to Gurobi but may assist in using the Gurobify package. In particular, it details functions which convert between characteristic vectors (which are more desirable for use with Gurobi) and sets (which may be more desirable for use in GAP).

3.9.1 IndexSetToCharacteristicVector (for IsList, IsPosInt)

▷ `IndexSetToCharacteristicVector(`*`IndexSet, NumberOfIndices`*`)` (operation)

Returns: Characterisitc vector

Takes a list of integers which form a subset of the set [1 .. n], where n is the second argument, and converts the set of indices to its characteristic vector. For example, if n = 5, the set [1,3] would be converted to [1, 0, 1, 0, 0]. It is useful to be able to convert between the two, since Gurobify always takes the characteristic vector (for example when taking constraints), yet the set of indices is generally more helpful for the user.

3.9.2 CharacteristicVectorToIndexSet (for IsList)

▷ `CharacteristicVectorToIndexSet(`*`CharacteristicVector`*`)` (operation)

Returns: Index set

Takes a characteristic vector and returns the set of indices corresponding to it. This reverses the process which occurs with IndexSetToCharacteristicVector. It is particularly useful to convert the output of a Gurobi solution back in terms of the variables. For example, the characteristic vector [1, 0, 1, 0, 0] would return the index set [1,3]. Note that since the function expects a characteristic vector it doesn't account for any weightings, and is only interested in whether or not the corresponding index is present, and as such it rounds each entry to the nearest integer and checks that it is non-zero. Hence it is particularly suitable for use with binary variables.

3.9.3 SubsetToCharacteristicVector (for IsList, IsList)

▷ `SubsetToCharacteristicVector(`*`Subset, FullSet`*`)` (operation)

Returns: Characterisitc vector

Takes a subset of some set, and returns the characteristic vector where the entries of the characteristic vector are indexed by the full set. For example, the subset ["c"] of ["a", "c", "n", "q"] would give the characteristic vector [0, 1, 0, 0]. This removes the need to first find the index set of the subset.

3.9.4 SubsetToIndexSet (for IsList, IsList)

▷ `SubsetToIndexSet(`*`Subset, FullSet`*`)` (operation)

Returns: Index set

Takes a subset of some set, and returns the set of indices corresponding to it. For example, the subset ["a", "d"] of the set ["a", "b", "c", "d", "e"], would return the index set [1,4]. Note that since the method expects a subset (not a multiset) vector it doesn't account for any weightings or repetition.

3.9.5 CharacteristicVectorToSubset (for IsList, IsList)

▷ `CharacteristicVectorToSubset(`*`CharacteristicVector`*`)` (operation)

Returns: Subset

Takes a characteristic vector and some set which it takes to be indexing the entries of the characteristic vector. It then returns the subset of the full set corresponding to the non-zero entries of the characteristic vector. This is the reverse process to SubsetToCharacteristicVector. Note again that the characteristic vector is rounded to an integer before being compared to 0. As an example, the characteristic vector [0, 1, 0, 0] with the set ["a", "c", "n", "q"] would return ["c"]. This removes the need to first return an index set before finding the subset.

3.10 Other

This section documents Gurobify functionality not belonging in any of the other sections.

3.10.1 GurobiVersion

▷ `GurobiVersion()` (function)

Returns: [major_version, minor_version, technical_version]

This function returns a list [major_version, minor_version, technical_version] which indicates the Gurobi library version. For example, if using Gurobi Version 7.5.1, then GurobiVersion() would return [7, 5, 1].

Chapter 4

Examples

This section contains a number of examples which are intended to illustrate the usage of Gurobify. These examples also form a test suite (along with "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));;
>          Add(var_names, name);
>        od;
>      od;
>    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 );
Gurobi model
```

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( var_names, var_included )
>      local vars, ind;
>      vars := ListWithIdenticalEntries( Size( var_names ), 0 );
>      ind := List( var_included, t -> Position( var_names, t ));
>      vars{ ind }:=ListWithIdenticalEntries( Size( var_included ), 1 );
>      return vars;
> end;
function( var_names, var_included ) ... end
gap> ExampleFuncIndexToNames := function( var_names, index_set )
>      local i, keep;
>      keep := [];
>      for i in var_names do
>         if index_set[Position( var_names, i)] = 1. then
>            Add(keep, i);
>         fi;
>      od;
>      return keep;
> end;
function( var_names, index_set ) ... end
gap>
gap> ExampleFuncDisplaySudoku := function( sol2 )
>      local mat_sol, m, i, j, k;
>        mat_sol := NullMat(9, 0);;
>        for m in sol2 do
>          i := EvalString( [m[2]] );
>          j := EvalString( [m[3]] );
>          k := EvalString( [m[4]] );
>          mat_sol[i][j] := k;
>        od;
>      Display(mat_sol);
>   end;
function( sol2 ) ... 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 :=[];
>          for k in [1 .. 9] do
>            name := Concatenation("x", String(i), String(j), String(k));;
>            Add(uniqueness_constr, name);
>          od;
```

```
>          constr := ExampleFuncNamesToIndex( var_names, uniqueness_constr );
>          GurobiAddConstraint( model, constr , "=", 1 );
>      od;
>   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 :=[];
>        for j in [1 .. 9] do
>          name := Concatenation("x", String(i), String(j), String(k));;
>          Add(row_constr, name );
>        od;
>        constr := ExampleFuncNamesToIndex( var_names, row_constr );
>        GurobiAddConstraint( model, constr, "=", 1 );
>      od;
>   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 :=[];
>        for i in [1 .. 9] do
>          name := Concatenation("x", String(i), String(j), String(k));;
>          Add(column_constr, name);
>        od;
>        constr := ExampleFuncNamesToIndex( var_names, column_constr );
>        GurobiAddConstraint(model, constr, "=", 1);
>      od;
>   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 := [];
>        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)
>               );;
>            Add(square_constr, name);
>          od;
>        od;
>        constr := ExampleFuncNamesToIndex( var_names, square_constr );
>        GurobiAddConstraint(model, constr, "=", 1);
>      od;
>   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 optimise. Change the solution into the set of variable names, and then display the solution.

Example

```
gap> GurobiOptimiseModel( model );
2
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 optimise the model and display the solution as before.

Example

```
gap> model2 := GurobiReadModel( "SudokuSolver.lp" );
Gurobi model
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> GurobiOptimiseModel(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 optimise. We feed this solution back in as a constraint, and then reoptimise. 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" );
Gurobi model
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> GurobiOptimiseModel( model3 );
2
gap> number_of_solutions := 0;;
gap> while GurobiOptimisationStatus( model3 ) = 2 do
>       sol := GurobiSolution( model3 );;
>       number_of_solutions := number_of_solutions + 1;
>       GurobiAddConstraint( model3, sol , "<", 80 );
>       GurobiUpdateModel( model3 );
>       GurobiReset(model3);
>       GurobiOptimiseModel( model3 );
> od;
gap> Print( number_of_solutions, "\n");
66
```

Chapter 5

Appendix

5.1 Gurobify Links

- Homepage: https://www.jesselansdown.com/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.

5.5 Troubleshooting The Installation

5.5.1 Typical Sources Of Error During Installation

Common sources of error during installation include:

- Forgetting to run ./install.sh
- Not having autotools installed. On a mac this can be done using brew, and checking all necessary dependencies are available with Brew Doctor.
- Not putting the correct paths to Gurobi (or GAP if not in the default "pkg" directory) when running "configure". Sometimes the paths are correct but incomplete, eg missing "linux64" or "mac64" at the end. Compare with the example paths given in the installation section of the manual.
- Having incorrect versions of GAP or Gurobi.

5.5.2 Specific Errors

- The compiler gives warnings when running "make"

This seems to be due to the way GAP stores integers, but it doesn't affect the function of the program. If no errors are given the installation is generally ok. Test the installation by loading Gurobify and running the test cases.

- Compilation is successful, but GAP cannot load the package, giving an error such as:

Example

```
gap> LoadPackage("Gurobify");
#W dlopen() error: libgurobi70.so: cannot open shared\
object file: No such file or directory
Error, module '/home/linux/gap4r8/pkg/Gurobify/bin\
/x86_64-pc-linux-gnu-gcc-default64/Gurobify.so'
```

This is caused by GAP not being able to find the Gurobi binary. When installing Gurobify, you were required to enter the path to Gurobi in the configure stage. If this path is "gurobi_path", then navigate to "gurobi_path/lib" in the terminal. You should now be able to load Gurobify when running GAP from this location. It can be fixed by running the following:

Example

```
LD_LIBRARY_PATH=gurobi_path/lib:$LD_LIBRARY_PATH
```

where "gurobi_path/lib" is as above. This will only be valid for the current session, to make this change permanent the line must be added to the .bash_profile file.

- Compilation is successful, but GAP cannot load Gurobify, giving an error such as:

Example

```
gap> LoadPackage("Gurobify");
dyld: lazy symbol binding failed: Symbol not found: _RegisterPackageTNUM
  Referenced from: /usr/local/lib/gap4r7/pkg/Gurobify-1.1.0/bin\
  /x86_64-apple-darwin14.3.0-gcc-default64/Gurobify.so
  Expected in: flat namespace
```

Gurobify requires GAP version 4.8 or higher.

- Gurobify will not compile, giving an error such as:

Example

```
src/Gurobify.c:12:10: fatal error: 'gurobi_c.h' file not found
```

Check that you have entered the correct path for Gurobi during installation.

- Gap gives an error like the following when trying to load Gurobify:

Example

```
gap> LoadPackage("Gurobify");
Error, Error: failed to create new environment. in
  if not LOAD_DYN( arg[1], false ) then
    Error( "no support for dynamic loading" );
fi; at /usr/local/lib/gap-4.10.0/lib/files.gd:583 called from
<function "LoadDynamicModule">( <arguments> )
 called from read-eval loop at /usr/local/lib/gap-4.10.0/pkg/Gurobify/init.g:8
type 'quit;' to quit to outer loop
```

Check that Gurobi's licese is up-to-date.

5.6 Example LP file

Shown below is the LP file, "test.lp", written during the simple example in Section 1.

Example

```
\ Model created with Gurobify - A GAP interface to Gurobi Optimizer.
\ LP format - for model browsing. Use MPS format to capture full model detail.
Minimize
  x + 2 y + z
Subject To
 UnNamedConstraint: 2 x + 2 y + 2 z <= 6
 UnNamedConstraint: x + 2 y + 3 z >= 5
 UnNamedConstraint: 2 x + 2 y + 2 z <= 6
 UnNamedConstraint: x + 2 y + 3 z >= 5
Bounds
Binaries
 x y z
End
```

References

[GAP16] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.8.6*, 2016. 5

[GH17] Sebastian Gutsche and Max Horn. *AutoDoc – a GAP package, Version 2017.09.15*, 2017. 2

[gur16a] Gurobi Optimization, Inc. *Gurobi Optimizer Reference Manual*, 2016. 5

[gur16b] Gurobi Optimizer Academic Program. http://www.gurobi.com/pdfs/Pricing-Academic.pdf, 2016. [Online; accessed 17-March-2017]. 6

[gur16c] Licensing Overview. http://www.gurobi.com/products/licensing-pricing/licensing-overview, 2016. [Online; accessed 17-March-2017]. 6

[gur16d] Optimization Status Codes. https://www.gurobi.com/documentation/7.0/refman/optimization_status_codes.html, 2016. [Online; accessed 21-March-2017]. 32

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

Index