Modeling Sudoku Puzzles with Python

Sean Davis, Matthew Henderson, Andrew Smith

Abstract—The popular Sudoku puzzles which appear daily in newspapers the world over have, lately, attracted the attention of mathematicians and computer scientists. There are many, difficult, unsolved problems about Sudoku puzzles and their generalizations which make them especially interesting to mathematicians. Also, as is well-known, the generalization of the Sudoku puzzle to larger dimension is an NP-complete problem and therefore of substantial interest to computer scientists.

In this article we discuss the modeling of Sudoku puzzles in a variety of different mathematical domains. We show how to use existing third-party Python libraries to implement these models. Those implementations, which include translations into the domains of constraint satisfaction, integer programming, polynomial calculus and graph theory, are available in an open-source Python library `sudoku.py` developed by the authors and available at http://bitbucket.org/matthew/scipy2010

Index Terms—sudoku, mathematics, graph theory

Introduction

Sudoku puzzles

A Sudoku puzzle is shown near the top of the second column on this page.

To complete this puzzle requires the puzzler to fill every empty cell with an integer between 1 and 9 in such a way that every number from 1 up to 9 appears once in every row, every column and every one of the small 3 by 3 boxes highlighted with thick borders.

Sudoku puzzles vary widely in difficulty. Determining the hardness of Sudoku puzzles is a challenging research problem for computational scientists. Harder puzzles typically have fewer prescribed symbols. However, the number of prescribed cells is not alone responsible for the difficulty of a puzzle and it is not well-understood what makes a particular Sudoku puzzle hard, either for a human or for an algorithm to solve.

The Sudoku puzzles which are published for entertainment invariably have unique solutions. A Sudoku puzzle is said to be *well-formed* if it has a unique solution. Another challenging research problem is to determine how few cells need to be filled for a Sudoku puzzle to be well-formed. Well-formed Sudoku with 17 symbols exist. It is unknown whether or not there exists a well-formed puzzle with only 16 clues. In this paper we consider all Sudoku puzzles, as defined in the next paragraph, not only the well-formed ones.

Sean Davis, Matthew Henderson, and Andrew Smith are with Berea College. E-mail: Sean_Davis@berea.edu.

2	5			3		9		1
	1				4			
4		7				2		8
		5	2					
				9	8	1		
	4				3			
			3	6			7	2
	7							3
9		3				6		4

By *Sudoku puzzle of boxsize n*, in this paper, is meant a partial assignment of values from $\{1,\ldots,n^2\}$ to the cells of an $n^2 \times n^2$ grid in such a way that at most one of each symbols occurs in any row, column or box. A *solution* of a Sudoku puzzle is a complete assignment to the cells, satisfying the same conditions on row, columns and boxes, which extends the original partial assignment.

sudoku.py

With `sudoku.py`, the process of building models of Sudoku puzzles, which can then be solved using algorithms for computing solutions of the models, is a simple matter. In order to understand how to build the models, first it is necessary to explain the two different representations of Sudoku puzzles in `sudoku.py`.

The dictionary representation of a puzzle is a mapping between cell labels and cell values. Cell values are integers in the range $\{1,\ldots,n^2\}$ and cell labels are integers in the range $\{1,\ldots,n^4\}$. The labeling of a Sudoku puzzle of boxsize n starts with 1 in the top-left corner and moves along rows, continuing to the next row when a row is finished. So, the cell in row i and column j is labeled $(i-1)n^2+j$.

For example, the puzzle from the introduction can be represented by the dictionary

```
>>> d = {1: 2, 2: 5, 5: 3, 7: 9, 9: 1,
...     11: 1, 15: 4, 19: 4, 21: 7, 25: 2,
...     27: 8, 30: 5, 31: 2, 41: 9, 42: 8,
...     43: 1, 47: 4, 51: 3, 58: 3, 59: 6,
...     62: 7, 63: 2, 65: 7, 72: 3, 73: 9,
...     75: 3, 79: 6, 81: 4}
```

A Sudoku puzzle object can be built from such a dictionary. Note that the boxsize is a parameter of the `Puzzle` object constructor.

```
>>> from sudoku import Puzzle
>>> p = Puzzle(d, 3)
>>> p
 2  5  .  .  3  .  9  .  1
 .  1  .  .  .  4  .  .  .
 4  .  7  .  .  .  2  .  8
 .  .  5  2  .  .  .  .  .
 .  .  .  .  9  8  1  .  .
 .  4  .  .  .  3  .  .  .
 .  .  .  3  6  .  .  7  2
 .  7  .  .  .  .  .  .  3
 9  .  3  .  .  .  6  .  4
```

In practice, however, the user mainly interacts with `sudoku.py` either by creating specific puzzles instances through input of puzzle strings, directly or from a text file, or by using generator functions.

The string representation of a Sudoku puzzle of boxsize n is a string of ascii characters of length n^4. In such a string a period character represents an empty cell and other ascii characters are used to specify assigned values. Whitespace characters and newlines are ignored when `Puzzle` objects are built from strings.

A possible string representation of the puzzle from the introduction is:

```
>>> s = """
... 2 5 . . 3 . 9 . 1
... . 1 . . . 4 . . .
... 4 . 7 . . . 2 . 8
... . . 5 2 . . . . .
... . . . . 9 8 1 . .
... . 4 . . . 3 . . .
... . . . 3 6 . . 7 2
... . 7 . . . . . . 3
... 9 . 3 . . . 6 . 4
    """
```

A `Puzzle` object can be built from a puzzle string by providing the keyword argument `format = 's'`

```
>>> p = Puzzle(s, 3, format = 's')
```

Random puzzles can be created in `sudoku.py` by the `random_puzzle` function.

```
>>> from sudoku import random_puzzle
>>> q = random_puzzle(15, 3)
>>> q
 .  .  .  .  5  .  .  .  1
 .  5  .  .  .  .  .  .  7
 .  .  1  9  .  7  .  .  .
 .  .  .  .  .  .  .  .  .
 .  .  5  .  .  .  7  .  .
 .  .  6  .  .  .  .  9  .
 .  .  .  .  .  5  .  .  .
 5  .  .  .  .  .  4  .  .
 1  .  .  .  .  .  .  .  .
```

The first argument to `random_puzzle` is the number of prescribed cells in the puzzle.

Solving of puzzles in `sudoku.py` is handled by the `solve` function. This function can use a variety of different algorithms, specified by an optional `model` keyword argument, to solve the puzzle. Possible values are `CP` for constraint propagation, `lp` for linear programming, `graph` to use a node coloring algorithm on a graph puzzle model and `groebner` to solve a polynomial system model via a Groebner basis algorithm. The default behavior is to use constraint propagation.

```
>>> from sudoku import solve
>>> s = solve(q)
>>> s
 7  3  2  8  5  6  9  4  1
 8  5  9  4  2  1  6  3  7
 6  4  1  9  3  7  8  5  2
 9  7  8  5  4  3  1  2  6
 3  2  5  6  1  9  7  8  4
 4  1  6  7  8  2  5  9  3
 2  9  4  1  6  5  3  7  8
 5  6  3  2  7  8  4  1  9
 1  8  7  3  9  4  2  6  5
```

Sudoku puzzles of boxsize other than 3 can also be modeled with `sudoku.py`. Puzzles of boxsize 2 are often called Shidoku.

```
>>> q2 = random_puzzle(7, 2)
>>> q2
 4  .  .  .
 2  1  .  .
 .  4  .  2
 .  .  3  4
>>> solve(q2)
 4  3  2  1
 2  1  4  3
 3  4  1  2
 1  2  3  4
```

Sudoku puzzles of boxsize greater than three are less commonly studied in the literature. In `sudoku.py` we use printable characters (from `string.printable`) for the symbols of puzzles with boxsize greater than 3

```
>>> q4 = random_puzzle(200, 4)
>>> q4
 .  .  e  d  .  .  a  9  8  .  .  5  .  3  2  1
 c  b  a  9  4  .  2  1  g  .  e  d  8  7  6  .
 8  .  6  5  g  f  e  d  4  3  2  1  c  b  a  9
 .  .  2  1  8  7  6  5  c  .  a  .  g  f  e  d
 f  d  g  .  9  8  7  c  3  6  .  b  .  2  .  .
 2  6  .  .  1  d  g  b  f  4  c  .  9  .  8  7
 .  4  1  8  3  6  .  2  9  e  7  .  .  .  5  c
 9  c  7  b  e  a  5  .  2  1  .  8  f  g  3  6
 e  g  9  f  7  .  8  a  6  d  3  4  5  1  b  .
 b  a  .  7  .  2  9  e  5  .  1  f  .  8  c  .
 3  8  .  6  5  1  4  f  .  9  b  2  7  a  d  g
 .  .  4  .  d  g  b  3  7  a  8  c  e  6  9  f
 .  e  f  c  2  9  3  8  a  5  g  7  6  4  .  b
 7  9  .  4  a  .  1  6  d  8  .  e  2  c  g  3
 6  2  8  g  b  .  d  .  .  c  9  3  .  .  f  .
 5  1  3  a  f  e  c  g  b  2  4  6  .  .  7  8
```

Solving puzzles of this size is still feasible by constraint propogation

```
>>> solve(q4)
g  f  e  d  c  b  a  9  8  7  6  5  4  3  2  1
c  b  a  9  4  3  2  1  g  f  e  d  8  7  6  5
8  7  6  5  g  f  e  d  4  3  2  1  c  b  a  9
4  3  2  1  8  7  6  5  c  b  a  9  g  f  e  d
f  d  g  e  9  8  7  c  3  6  5  b  1  2  4  a
2  6  5  3  1  d  g  b  f  4  c  a  9  e  8  7
a  4  1  8  3  6  f  2  9  e  7  g  b  d  5  c
```

```
9  c  7  b  e  a  5  4  2  1  d  8  f  g  3  6
e  g  9  f  7  c  8  a  6  d  3  4  5  1  b  2
b  a  d  7  6  2  9  e  5  g  1  f  3  8  c  4
3  8  c  6  5  1  4  f  e  9  b  2  7  a  d  g
1  5  4  2  d  g  b  3  7  a  8  c  e  6  9  f
d  e  f  c  2  9  3  8  a  5  g  7  6  4  1  b
7  9  b  4  a  5  1  6  d  8  f  e  2  c  g  3
6  2  8  g  b  4  d  7  1  c  9  3  a  5  f  e
5  1  3  a  f  e  c  g  b  2  4  6  d  9  7  8
```

Models

In this section we introduce several models of Sudoku and show how to use existing Python components to implement these models. The models introduced here are all implemented in `sudoku.py`. Implementation details are discussed in this section and demonstrations of the components of `sudoku.py` corresponding to each of the different models are given.

Constraint models

Constraint models for Sudoku puzzles are discussed in [Sim05]. A simple model uses the AllDifferent constraint.

A constraint program is a collection of constraints. A constraint restricts the values which can be assigned to certain variables in a solution of the constraint problem. The AllDifferent constraint restricts variables to having mutually different values.

Modeling Sudoku puzzles is easy with the AllDifferent constraint. To model the empty Sudoku puzzle (i.e. the puzzle with no clues) a constraint program having an AllDifferent constraint for every row, column and box is sufficient.

For example, if we let $x_i \in \{1,\ldots,n^2\}$ for $1 \leq i \leq n^4$, where $x_i = j$ means that cell i gets value j then the constraint model for a Sudoku puzzle of boxsize $n = 3$ would include constraints:

$$\text{AllDifferent}(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9)$$

$$\text{AllDifferent}(x_1, x_{10}, x_{19}, x_{28}, x_{37}, x_{46}, x_{55}, x_{64}, x_{73})$$

$$\text{AllDifferent}(x_1, x_2, x_3, x_{10}, x_{11}, x_{12}, x_{19}, x_{20}, x_{21})$$

These constraints ensure that, respectively, the variables in the first row, column and box get different values.

The Sudoku constraint model in `sudoku.py` is implemented using `python-constraint v1.1` by Gustavo Niemeyer. This open-source library is available at http://labix.org/python-constraint.

With `python-constraint` a `Problem` having variables for every cell $\{1,\ldots,n^4\}$ of the Sudoku puzzle is required. The list of cell labels is given by the function `cells` in `sudoku.py`. Every variable has the same domain $\{1,\ldots,n^2\}$ of symbols. The list of symbols in `sudoku.py` is given by the `symbols` function.

The `Problem` member function `addVariables` provides a convenient method for adding variables to a constraint problem object.

```
>>> from constraint import Problem
>>> from sudoku import cells, symbols
>>> cp = Problem()
>>> cp.addVariables(cells(n), symbols(n))
```

The AllDifferent constraint in `python-constraint` is implemented as `AllDifferentConstraint()`. The `addConstraint(constraint, variables)` member function is used to add a `constraint` on `variables` to a constraint `Problem` object. So, to build an empty Sudoku puzzle constraint model we can do the following.

```
>>> from constraint import AllDifferentConstraint
>>> from sudoku import \
...   cells_by_row, cells_by_col, cells_by_box
>>> for row in cells_by_row(n):
...   cp.addConstraint(AllDifferentConstraint(), row)
>>> for col in cells_by_col(n):
...   cp.addConstraint(AllDifferentConstraint(), col)
>>> for box in cells_by_box(n):
...   cp.addConstraint(AllDifferentConstraint(), box)
```

Here the functions `cells_by_row`, `cells_by_col` and `cells_by_box` give the cell labels of a Sudoku puzzle ordered, respectively, by row, column and box. These three loops, respectively, add to the constraint problem object the necessary constraints on row, column and box variables.

To extend this model to a Sudoku puzzle with clues requires additional constraints to ensure that the values assigned to clue variables are fixed. One possibility is to use an ExactSum constraint for each clue.

The ExactSum constraint restricts the sum of a set of variables to a precise given value. We can slightly abuse the ExactSum constraint to specify that certain individual variables are given certain specific values. In particular, if the puzzle clues are given by a dictionary `d` then we can complete our model by adding the following constraints.

```
>>> from constraint import ExactSumConstraint as Exact
>>> for cell in d:
...   cp.addConstraint(Exact(d[cell]), [cell])
```

To solve the Sudoku puzzle now can be done by solving the constraint model `cp`. The constraint propogation algorithm of `python-constraint` can be invoked by the `getSolution` member function.

```
>>> s = Puzzle(cp.getSolution(), 3)
>>> s
 2  5  8  7  3  6  9  4  1
 6  1  9  8  2  4  3  5  7
 4  3  7  9  1  5  2  6  8
 3  9  5  2  7  1  4  8  6
 7  6  2  4  9  8  1  3  5
 8  4  1  6  5  3  7  2  9
 1  8  4  3  6  9  5  7  2
 5  7  6  1  4  2  8  9  3
 9  2  3  5  8  7  6  1  4
```

The general `solve` function of `sudoku.py` knows how to build the constraint model above, find a solution via the propogation algorithm of `python-constraint` and translate the solution into a completed Sudoku puzzle.

```
>>> s = solve(p, model = 'CP')
```

Here, `p` is a `Puzzle` instance. In fact, the `model = 'CP'` keyword argument in this case is redundant, as `'CP'` is the default value of `model`.

Graph models

A graph model for Sudoku is presented in [Var05]. In this model, every cell of the Sudoku grid is represented by a

node of the graph. The edges of the graph are given by the dependency relationships between cells. In other words, if two cells lie in the same row, column or box, then their nodes are joined by an edge in the graph.

In the graph model, a Sudoku puzzle is given by a partial assignment of colors to the nodes of the graph. The color assigned to a node corresponds to a value assigned to the corresponding cell. A solution of the puzzle is given by a coloring of the nodes with colors $\{1,\ldots,n^2\}$ which extends the original partial coloring. A node coloring of the Sudoku graph which corresponds to a completed puzzle has the property that adjacent vertices are colored differently. Such a node coloring is called *proper*.

The Sudoku graph model in `sudoku.py` is implemented using `networkx v1.1`. This open-source Python graph library is available at http://networkx.lanl.gov/

Modeling an empty Sudoku puzzle as a `networkx.Graph` object requires nodes for every cell and edges for every pair of dependent cells. To add nodes (respectively, edges) to a graph, `networkx` provides member functions `add_nodes_from` (respectively, `add_edges_from`). Cell labels are obtained from `sudoku.py`'s `cells` function.

```
>>> import networkx
>>> g = networkx.Graph()
>>> g.add_nodes_from(cells(n))
```

Dependent cells are computed using the `dependent_cells` function. This function returns the list of all pairs (x,y) with $x<y$ such that x and y either lie in the same row, same column or same box.

```
>>> from sudoku import dependent_cells
>>> g.add_edges_from(dependent_cells(n))
```

To model a Sudoku puzzle, we have to be able to assign colors to nodes. Graphs in `networkx` allow arbitrary data to be associated with graph nodes. To color nodes according to the dictionary `d` of puzzle clues.

```
>>> for cell in d:
...   g.node[cell]['color'] = d[cell]
```

There are many node coloring algorithms which can be used to find a solution of a puzzle. In `sudoku.py`, a generic node coloring algorithm is implemented. This generic coloring algorithm can be customized to provide a variety of different specific coloring algorithms. However, none of these algorithms is guaranteed to find a soolution which uses only symbols from $\{1,\ldots,n^2\}$. In general, these algorithms use too many colors

```
>>> from sudoku import node_coloring, n_colors
>>> cg = node_coloring(g)
>>> n_colors(cg)
13
>>> from sudoku import graph_to_dict
>>> s = Puzzle(graph_to_dict(cg), 3)
>>> s
 2  5  6  7  3  a  9  4  1
 3  1  8  5  2  4  7  6  a
 4  9  7  6  b  c  2  3  8
 6  3  5  2  4  7  8  9  b
 7  2  a  b  9  8  1  5  6
 8  4  9  a  5  3  c  2  7
 5  8  4  3  6  9  a  7  2
 a  7  b  4  8  5  d  c  3
 9  c  3  d  7  b  6  8  4
```

To solve a Sudoku `Puzzle` instance `p`, call the `solve` function, with `model = graph` as a keyword argument.

```
>>> s = solve(p, model = 'graph')
```

Polynomial system models

The graph model above is introduced in [Var05] as a prelude to modeling Sudoku puzzles as systems of polynomial equations. The polynomial system model in [Var05] involves variables x_i for $i \in \{1,\ldots,n^4\}$ where $x_i = j$ is interpreted as the cell with label i being assigned the value j.

The Sudoku polynomial-system model in sudoku.py is implemented using `sympy v0.6.7`. This open-source symbolic algebra Python library is available at http://code.google.com/p/sympy/

Variables in `sympy` are `Symbol` objects. A `sympy.Symbol` object has a name. So, to construct the variables for our model, first we map symbol names onto each cell label.

```
>>> from sudoku import cell_symbol_name
>>> def cell_symbol_names(n):
...   return map(cell_symbol_name, cells(n))
```

Now, with these names for the symbols which represent the cells of our Sudoku puzzle, we can construct the cell variable symbols themselves.

```
>>> from sympy import Symbol
>>> def cell_symbols(n):
...   return map(Symbol, cell_symbol_names(n))
```

Finally, with these variables, we can build a Sudoku polynomial system model. This model is based on the graph model of the previous section. There are polynomials in the system for every node in the graph model and polynomials for every edge.

The role of node polynomial $F(x_i)$ is to ensure that every cell i is assigned a number from $\{1,\ldots,n^2\}$:

$$F(x_i) = \prod_{j=1}^{n^2}(x_i - j)$$

Node polynomials, for a `sympy.Symbol` object `x` are built as follows.

```
>>> from operator import mul
>>> from sudoku import symbols
>>> def F(x,n):
...   return reduce(mul,[(x-s) for s in symbols(n)])
```

The edge polynomial $G(x_i,x_j)$ for dependent cells i and j, ensures that cells i and j are assigned different values. These polynomials have the form. :

$$G(x_i,x_j) = \frac{F(x_i) - F(x_j)}{x_i - x_j}$$

In `sympy`, we build edge polynomials from the node polynomial function `F`.

```
>>> from sympy import cancel, expand
>>> def G(x,y,n):
...   return expand(cancel((F(x,n)-F(y,n))/(x-y)))
```

The polynomial model for the empty Sudoku puzzle consists of the collection of all node polynomials for nodes in the Sudoku graph and all edge polynomials for pairs `(x,y)` in `dependent_symbols(n)`. The `dependent_symbols` function is simply a mapping of the `sympy.Symbol` constructor onto the list of dependent cells.

Specifying a Sudoku puzzle requires extending this model by adding polynomials to represent clues. According to the model from [Var05], if D is the set of fixed cells (i.e. cell label, value pairs) then to the polynomial system we need to add polynomials

$$D(x_i, j) = x_i - j$$

Or, with `sympy`:

```
>>> def D(i, j):
...   return Symbol(cell_symbol_name(i)) - j
```

To build the complete polynomial system, use the `puzzle_as_polynomial_system` function of `sudoku.py`:

```
>>> from sudoku import puzzle_as_polynomial_system
>>> g = puzzle_as_polynomial_system(d, 3)
```

The `sympy` implementation of a Groebner basis algorithm can be used to find solutions of this polynomial system. The Groebner basis depends upon a variable ordering, here specified as lexicographic. Other orderings, such as degree-lexicographic, are possible.

```
>>> from sympy import groebner
>>> h = groebner(g, cell_symbols(n), order = 'lex')
```

The solution of the polynomial system `g` is a system of linear equations in the symbols x_i which can be solved by the linear solver from `sympy`.

```
>>> from sympy import solve as lsolve
>>> s = lsolve(h, cell_symbols(n))
```

To use the polynomial-system model to find a solution to `Puzzle` instance `p` call the solve function with the keyword argument `model = groebner`.

```
>>> s = solve(p, model = 'groebner')
```

Integer programming models

In [Bar08] a model of Sudoku as an integer programming problem is presented. In this model, the variables are all binary.

$$x_{ijk} \in \{0, 1\}$$

Variable x_{ijk} represents the assignment of symbol k to cell (i, j) in the Sudoku puzzle.

$$x_{ijk} = \begin{cases} 1 & \text{if cell } (i,j) \text{ contains symbol } k \\ 0 & \text{otherwise} \end{cases}$$

The integer programming (IP) model has a set of equations which force the assignment of a symbol to every cell.

$$\sum_{k=1}^{n} x_{ijk} = 1, \quad 1 \le i \le n, 1 \le j \le n$$

Other equations in the IP model represent the unique occurence of every symbol in every column:

$$\sum_{i=1}^{n} x_{ijk} = 1, \quad 1 \le j \le n, 1 \le k \le n$$

every symbol in every row:

$$\sum_{j=1}^{n} x_{ijk} = 1, \quad 1 \le i \le n, 1 \le k \le n$$

and every symbol in every box:

$$\sum_{j=mq-m+q}^{mq} \sum_{i=mp-m+1}^{mp} x_{ijk} = 1$$

$$1 \le k \le n, 1 \le p \le m, 1 \le q \le m$$

The Sudoku IP model is implemented in `sudoku.py` using `pyglpk v0.3` by Thomas Finley. This open-source mixed integer/linear programming Python library is available at http://tfinley.net/software/pyglpk/

In `pyglpk`, an integer program is represented by the matrix of coefficients of a system of linear equations. Two functions of `sudoku.py` provide the correct dimensions of the coefficient matrix.

```
>>> from glpk import LPX
>>> from sudoku import \
...   lp_matrix_ncols, lp_matrix_nrows
>>> lp = LPX()
>>> lp.cols.add(lp_matrix_ncols(n))
>>> lp.rows.add(lp_matrix_nrows(n))
```

Columns of the matrix represent different variables. All our variables are binary and so their bounds are set appropriately, between 0 and 1.

```
>>> for c in lp.cols:
...   c.bounds = 0.0, 1.0
```

Rows of the coefficient matrix represent different linear equations. We require all our equations to have a value of 1, so we set both the lower and upper bound of every equation to be 1.

```
>>> for r in lp.rows:
...   r.bounds = 1.0, 1.0
```

With appropriate dimensions and bounds fixed, the coefficient matrix itself is provided by `sudoku.py`'s `lp_matrix` function.

```
>>> from sudoku import lp_matrix
>>> lp.matrix = lp_matrix(n)
```

To extend the IP model to a Sudoku puzzle with fixed clues requires further equations. Fixed elements in the puzzle, given by a set F of triples (i, j, k), are each represented by an equation in the system:

$$x_{ijk} = 1, \quad \forall (i, j, k) \in F$$

To add these equations to the `pyglpk.LPX` object `lp`:

```
>>> from sudoku import lp_col_index
>>> for cell in d:
...   lp.rows.add(1)
...   r = lp_matrix_ncols(n)*[0]
...   r[lp_col_index(cell, d[cell], n)] = 1
...   lp.rows[-1].matrix = r
...   lp.rows[-1].bounds = 1.0, 1.0
```

To solve the `LPX` instance `lp` requires first solving a linear relaxation via the simplex algorithm implementation of `pyglpk`

```
>>> lp.simplex()
```

Once the linear relaxation is solved, the original integer program can be solved.

```
>>> for col in lp.cols:
...   col.kind = int
>>> lp.integer()
```

Finally, we need to extract the solution as a dictionary from the model via the `lp_to_dict` function from `sudoku.py`.

```
>>> from sudoku import lp_to_dict
>>> d = lp_to_dict(lp, n)
>>> s = Puzzle(d, 3)
>>> s
 2  5  8  7  3  6  9  4  1
 6  1  9  8  2  4  3  5  7
 4  3  7  9  1  5  2  6  8
 3  9  5  2  7  1  4  8  6
 7  6  2  4  9  8  1  3  5
 8  4  1  6  5  3  7  2  9
 1  8  4  3  6  9  5  7  2
 5  7  6  1  4  2  8  9  3
 9  2  3  5  8  7  6  1  4
```

To use the IP model to solve a `Puzzle` instance, specify the keyword argument `model = lp`.

```
>>> s = solve(p, model = 'lp')
```

Experimentation

In this section we demonstrate the use of `sudoku.py` for creating Python scripts for experimentation with Sudoku puzzles. For the purposes of demonstration, we discuss, briefly, enumeration of Shidoku puzzles, coloring the Sudoku graph and the hardness of random puzzles.

Enumerating Shidoku

Enumeration of Sudoku puzzles is a very difficult computational problem, which has been solved by Felgenhauer and Jarvis in [Fel06]. The enumeration of Shidoku, however, is easy. To solve the enumeration problem for Shidoku, using the constraint model implemented in `sudoku.py`, takes only a few lines of code and a fraction of a second of computation.

```
>>> s = "from sudoku import Puzzle, count_solutions"
>>> e = "print count_solutions(Puzzle({}, 2))"
>>> from timeit import Timer
>>> t = Timer(e, s)
>>> print t.timeit(1)
288
0.146998882294
```

Coloring the Sudoku graph

As discussed above in the section on "Graph models", a completed Sudoku puzzle is equivalent to a minimal proper node coloring of the Sudoku graph. We have experimented with several different node coloring algorithms to see which are more effective, with respect to minimizing the number of colors, at coloring the Sudoku graph.

Initially, we used Joseph Culberson's graph coloring programs (http://webdocs.cs.ualberta.ca/~joe/Coloring/index.html) by writing Sudoku puzzle graphs to a file in Dimacs format (via the `dimacs_string` function of `sudoku.py`).

Of those programs we experimented with, the program implementing the saturation degree algorithm (DSatur) of Brelaz from [Bre79] seemed most effective at minimizing the number of colors.

Motivated to investigate further, with `sudoku.py` we implemented a general node coloring algorithm directly in Python which can reproduce the DSatur algorithm as well as several other node coloring algorithms.

Our node coloring function allows for customization of a quite general scheme. The behavior of the algorithm is specialized by two parameters. The `nodes` parameter is an iterable object giving a node ordering. The `choose_color` parameter is a visitor object which is called every time a node is visited by the algorithm.

Several node orderings and color choice selection schemes have been implemented. The simplest sequential node coloring algorithm can be reproduced, for example, by assigning `nodes = InOrder` and `choose_color = first_available_color`. A random ordering on nodes can be acheived instead by assigning `nodes = RandomOrder`. Importantly for our investigations, the node ordering is given by an iterable object and so, in general, can reflect upon to current graph state. This mean that online algorithms like the DSatur algorithm can be realized by our general node coloring scheme. The DSatur algorithm is obtained by assigning `nodes = DSATOrder` and `choose_color = first_available_color`.

Hardness of random puzzles

We introduced the `random_puzzle` function in the introduction. The method by which this function produces a random puzzle is fairly simple. A completed Sudoku puzzle is first generated by solving the empty puzzle via constraint propagation and then from this completed puzzle the appropriate number of clues is removed.

An interesting problem is to investigate the behavior of different models on random puzzles. A simple script, available in the `investigations` folder of the source code, has been written to time the solution of models of random puzzles and plot the timings via `matplotlib`.

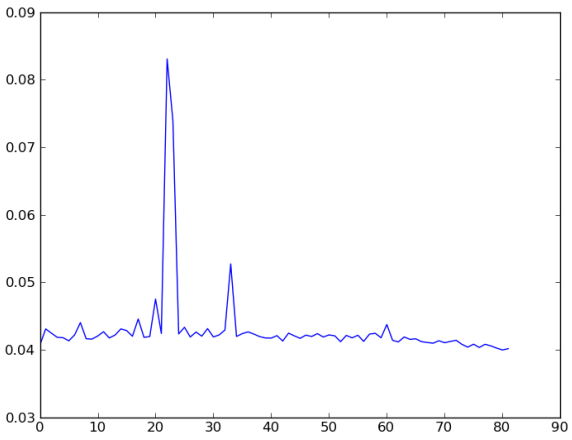
Two plots produced by this script highlight the different behavior of the constraint model and the integer programming model.

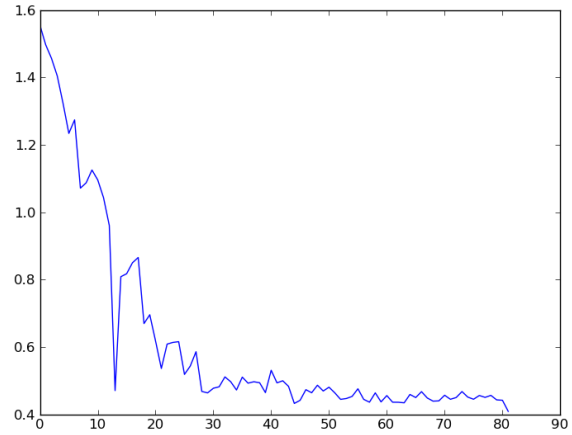
The first plot has time on the vertical axis and the number of clues on the horizontal axis. From this plot it seems that the constraint propogation algorithm finds puzzles with many or few clues easy. The difficult problems for the constraint solver appear to be clustered in the range of 20 to 35 clues.

A different picture emerges with the linear programming model. With the same set of randomly generated puzzles it appears that the more clues the faster the solver finds a solution.

Conclusions and future work

In this article we introduced `sudoku.py`, an open-source Python library for modeling Sudoku puzzles. We discussed several models of Sudoku puzzles and demonstrated how to implement these models using existing Python libraries. A few simple experiments involving Sudoku puzzles were presented.





Future plans for `sudoku.py` are to increase the variety of models. Both by allowing for greater customization of currently implemented models and by implementing new models. For example, we can imagine several different Sudoku models as constraint programs beyond the model presented here. Another approach is to model Sudoku puzzles as exact cover problems and investigate the effectiveness of Knuth's dancing links algorithm. Also important to us is to compare all our models with models [Lyn06] from satisfiability theory. In [Kul10] a general scheme is presented which is highly effective for modeling Sudoku.

There are great many interesting, unsolved scientific problems involing Sudoku puzzles. Our hope is that `sudoku.py` can become a useful tool for scientists who work on these problems.

REFERENCES

[Bar08] A. Bartlett, T. Chartier, A. Langville, T. Rankin. *An Integer Programming Model for the Sudoku Problem*, J. Online Math. & Its Appl., 8(May 2008), May 2008

[Bre79] Brelaz, D., *New methods to color the vertices of a graph*, Communications of the Assoc. of Comput. Machinery 22 (1979), 251-256.

[Fel06] B. Felgenhauer, F. Jarvis. *Enumerating possible Sudoku grids* Online resource 2006 http://www.afjarvis.staff.shef.ac.uk/sudoku/

[Kul10] O. Kullmann, *Green-Tao numbers and SAT* in LNCS (Springer), "Theory and Applications of Satisfiability Testing - SAT 2010", editors O. Strichman and S. Szeider

[Lew05] R. Lewis. *Metaheuristics can solve Sudoku puzzles*, Journal of Heuristics (2007) 13: 387-401

[Lyn06] Lynce, I. and Ouaknine. *Sudoku as a SAT problem*, Proceedings of the 9th Symposium on Artificial Intelligence and Mathematics, 2006.

[Sim05] H. Simonis. *Sudoku as a Constraint Problem*, Proceedings of the 4th International Workshop on Modelling and Reformuulating Constraint Satisfaction Problems. pp.13-27 (2005)

[Var05] J. Gago-Vargas, I. Hartillo-Hermosa, J. Martin-Morales, J. M. Ucha-Enriquez, *Sudokus and Groebner Bases: not only a Divertimento*, In: Lecture Notes in Computer Science, vol. 4194. pp. 155-165. 2005