Homework 16

1 Homework 16 (PRECHECKED)

Problem

Create a linear program to solve a sudoku problem.

Context

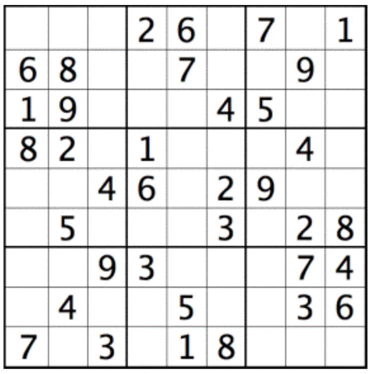
A sudoku puzzle is traditionally a 9x9 grid with some numbers provided in seemingly random cells. Solving a sudoku problems involves filling out all of the remaining cells with the given conditions:

1. Every column in the grid must contain every integer from 1 to 9.
2. Every row in the grid must contain every integer from 1 to 9.
3. The grid is split into 9 smaller 3x3 sections, and each of those subgrids must contain every integer from 1 to 9.
4. No duplicate values may exist in any column, row or subgrid.

Solution

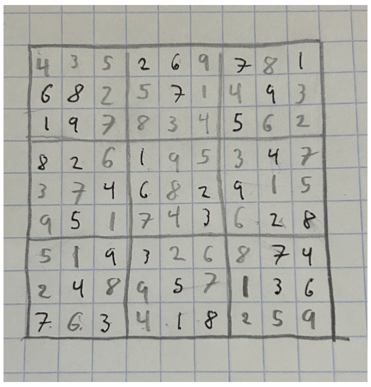
I leveraged two different solutions to this problem. The first involved using a one-hot encoding strategy for handling the sudoku grid. The second leveraged the `alldiff` function that some solvers have access to. I will be covering both here.

Below is the sudoku problem that I used as a base for this homework.



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

And here is my solution for the problem done by hand.



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

The Data File

Here is how I set up the above sudoku problem. I will not include the full file here, the snippet here should suffice.

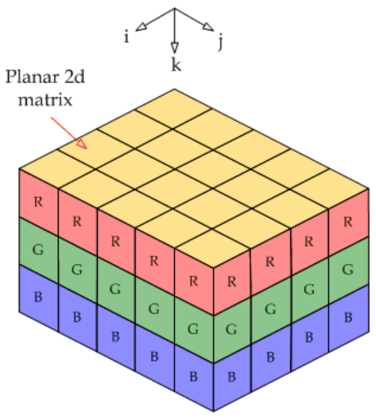
```
data;

param gridsize := 9;

param grid_data :=
1 4 2
1 5 6
1 7 7
1 9 1
2 1 6
2 2 8
2 5 7
...
```

I organize the `grid_data` as “row, column, value”. So in this case row 1 column 4 has the value 2. I manually filled in all given values this way.

Solution A: One-hot Encoding

I had originally wanted to think of this as a knapsack problem but got overwhelmed trying to track everything. This solution, instead of limiting itself to a 2d grid, expands out to a cube, with a matrix for each possible value.

We can think of this much like an image with different channels for RGB. Each pixel in an image can have values for each color. Below is a visualization of this.



Graphical presentation of RGB 3d matrix

So in our case we have 9 9x9 grids. One for each possible value. It’s the same sudoku grid stacked on top of each other, and each layer simply asks “does this cell have value x in it”? This seems bizarre at first but when we add constraints into the mix we get a very convenient structure for finding a solution. For starters, we can say “this cell can only contain red, green or blue”. So if we’d put 2 in a cell, the 2 layer gets a 1, or a “True”, in the corresponding cell. Every other layer would get a 0 in that cell.

This expands out in a very convenient way. This allows us to easily use sums for ALL checks which is a mess when you're working with the values themselves. Let's walk through the model now.

MODEL CODE

First we set up our grid in 2D. We want to read in the data and track everything that's been given. We want to make sure none of those can be modified later.

```
param gridsize > 0;
param grid_data {1..gridsize, 1..gridsize} integer >= 0, < 10, default 0;

# differentiates between given cells and cells we can modify
set FIXED := {i in 1..gridsize, j in 1..gridsize: grid_data[i,j] != 0};
set MODIFIABLE := {i in 1..gridsize, j in 1..gridsize: grid_data[i,j] = 0};
```

Now here is where we set up our cube. The cube itself is our variable for the linear program.

```
# possible values
set VALS := 1..9;

# Like a binary RGB grid for images. Asks "Does this cell have red?"
var X {i in 1..gridsize, j in 1..gridsize, v in VALS} binary;
```

Next we get to the constraints. How do we want the cube to behave?

```
# Restrict the models ability to change any given values
subject to prefilled { (i,j) in FIXED }:
    X[i,j, grid_data[i,j]] = 1;

# forces every cell to HAVE a value
# One of those value windows has to have something
subject to one_value_per_cell { (i,j) in MODIFIABLE }:
    sum {v in VALS} X[i,j,v] = 1;
```

The above constraint is really important, without it we can satisfy the other conditions and end up with a bunch of 0 cells. So every cell must have one and only one active layer. Next up, the classic constratints.

```
subject to row_unique {i in 1..gridsize, v in VALS}:
    sum {j in 1..gridsize} X[i,j,v] = 1;

subject to col_unique {j in 1..gridsize, v in VALS}:
    sum {i in 1..gridsize} X[i,j,v] = 1;

subject to block_unique {bi in 0..2, bj in 0..2, v in VALS}:
    sum {i in 3*bi+1..3*bi+3, j in 3*bj+1..3*bj+3} X[i,j,v] = 1;
```

These enforce the typical sudoku rules.

Now the objective function in this case is totally arbitrary. I just choose to sum everything up.

```
# Totally arbitrary objective function, just sum up everything
maximize dummy: sum {i in 1..gridsize, j in 1..gridsize, v in VALS} X[i,j,v];
```

OUTPUT

The output is huge here so I'll only show the relevant snippets and overall solution.

Here is our given problem

```
grid_data [*,*]
:   1   2   3   4   5   6   7   8   9    :=
1   0   0   0   2   6   0   7   0   1
2   6   8   0   0   7   0   0   9   0
3   1   9   0   0   0   4   5   0   0
4   8   2   0   1   0   0   0   4   0
5   0   0   4   6   0   2   9   0   0
6   0   5   0   0   0   3   0   2   8
7   0   0   9   3   0   0   0   7   4
8   0   4   0   0   5   0   0   3   6
9   7   0   3   0   1   8   0   0   0
```

Here is the solution (cleaned up with python)

```
[4, 3, 5, 2, 6, 9, 7, 8, 1]
[6, 8, 2, 5, 7, 1, 4, 9, 3]
[1, 9, 7, 8, 3, 4, 5, 6, 2]
[8, 2, 6, 1, 9, 5, 3, 4, 7]
[3, 7, 4, 6, 8, 2, 9, 1, 5]
[9, 5, 1, 7, 4, 3, 6, 2, 8]
[5, 1, 9, 3, 2, 6, 8, 7, 4]
[2, 4, 8, 9, 5, 7, 1, 3, 6]
[7, 6, 3, 4, 1, 8, 2, 5, 9]
```

And to show an example of the makeup of this solution under the hood, here is the layer for the value 1. This will show all the locations there the cells is 1.

```
X[i,j,1] [*,*]
:   1   2   3   4   5   6   7   8   9    :=
1   0   0   0   0   0   0   0   0   1
2   0   0   0   0   0   1   0   0   0
3   1   0   0   0   0   0   0   0   0
4   0   0   0   1   0   0   0   0   0
5   0   0   0   0   0   0   0   1   0
6   0   0   1   0   0   0   0   0   0
7   0   1   0   0   0   0   0   0   0
8   0   0   0   0   0   0   1   0   0
9   0   0   0   0   1   0   0   0   0
```

Solution B - alldiff

I'll keep this section brief as it is far simpler. Here we use the power of alternative solvers like `scip-cpx` that allows for a powerful constraint tool, `alldiff`. This forces a set of integer variables to have distinct values. Using this with a set of possible values, say `1..9` makes building a model extremely easy.

MODEL CODE

For this solution we read in the data and track our fixed and modifiable indices in the exact same way. The differences are shown here.

```
var X {i in 1..gridsize, j in 1..gridsize} integer >= 1, <= 9;

# Restrict the models ability to change any given values
subject to prefilled { (i,j) in FIXED }:
    X[i,j] = grid_data[i,j];

subject to row_unique {i in 1..gridsize}:
    alldiff {j in 1..gridsize} X[i,j];

subject to col_unique {j in 1..gridsize}:
    alldiff {i in 1..gridsize} X[i,j];

subject to block_unique {bi in 0..2, bj in 0..2}:
    alldiff {i in 3*bi+1..3*bi+3, j in 3*bj+1..3*bj+3} X[i,j];
```

We use the exact same constraints as before but instead of summing across all the layers we just slap on the `alldiff` constraint. This allows us to stay with just a 2d matrix the entire time.

OUTPUT

```
X [*,*]
:   1   2   3   4   5   6   7   8   9    :=
1   4   3   5   2   6   9   7   8   1
2   6   8   2   5   7   1   4   9   3
3   1   9   7   8   3   4   5   6   2
4   8   2   6   1   9   5   3   4   7
5   3   7   4   6   8   2   9   1   5
6   9   5   1   7   4   3   6   2   8
7   5   1   9   3   2   6   8   7   4
8   2   4   8   9   5   7   1   3   6
9   7   6   3   4   1   8   2   5   9
```

As we cam see it reaches the same solution as both myself and the previous model.

Some fun notes!

Sudoku is considered an **NP-Complete** problem. This means a few things.

1. The problem has a decision (yes/no) structure. “Can we fill the sudoku grid such that…”
2. There currently isn’t a known way to solve a sudoku problem in polynomial time.
3. Verifying a sudoku solution **can** be done in polynomial time.
4. If a polynomial time solution was found for sudoku, that solution could then be used to find a polynomial time solution to all other NP Complete problems.

Some examples:

- Sudoku (duh)
- Graph Coloring
- Knapsack Problem (decision version)

- Traveling Salesman