CS24011: Lab 2: Constraint Satisfaction

Ian Pratt-Hartmann
Francisco Lobo

Academic session: 2022-23

Git Tag: `24011-lab2-S-constraints`

Introduction

This Laboratory features the constraint satisfaction solver implemented in the Python `constraint` module by Gustavo Niemeyer. It consists of two exercises.

- In the first, you will solve the 'logic puzzle' encountered in the introductory lecture to this course (the one about the travellers arriving at an airport). The purpose of this exercise is to practice turning a problem expressed informally in English into a constraint satisfaction problem (CSP). If you are successful, you will see that the Python constraint satisfaction package works very well.
- In the second exercise, we apply the same constraint satisfaction package to a more abstract problem in recreational mathematics. The problem in question is that of generating pan-diagonal magic squares. (We explain what these are below.) The purpose of this exercise is to illustrate some of the limitations of CSP solvers — especially when the problem instances they are given grow in size.

Documentation for the `constraint` module is provided at

`https://labix.org/python-constraint`

Try this out. First, you will need to install the Python `constraint` module. It should suffice to type into your command shell

```
$ pip install python-constraint
```

The logic puzzle

Let us recall the logic puzzle encountered in the introductory lecture. Four travellers — Claude, Olga, Pablo and Scott — are returning from four countries — Peru, Romania, Taiwan and Yemen — departing at four different times — 2:30, 3:30, 4:30 and 5:50 (one hopes: in the afternoon). No two travellers return from the same destination, and no two depart at the same time. We are given the following *special constraints*:

1. Olga is leaving 2 hours earlier than the traveller flying from Yemen.
2. Claude is either the person leaving at 2:30 pm or the traveller leaving at 3:30 pm.
3. The person leaving at 2:30 pm is flying from Peru.
4. The person flying from Yemen is leaving earlier than the person flying from Taiwan.
5. The four travellers are Pablo, the traveller flying from Yemen, the person leaving at 2:30 pm and the person leaving at 3:30 pm.

We need to set this up using the Python `constraint` module. Here is how I did it:

```python
from constraint import Problem, AllDifferentConstraint
problem= Problem()

people= ["claude", "olga", "pablo", "scott"]
times= ["2:30", "3:30", "4:30", "5:30"]
destinations= ["peru", "romania", "taiwan", "yemen"]

t_variables= list(map(lambda x: "t_"+x, people))
d_variables= list(map(lambda x: "d_"+x, people))

problem.addVariables(t_variables, times)
problem.addVariables(d_variables, destinations)

problem.addConstraint(AllDifferentConstraint(),t_variables)
problem.addConstraint(AllDifferentConstraint(),d_variables)
```

This code should be self-explanatory. Note the clever short-cut to say that certain sets of variables have to have different values. (It would be annoying to have to write this out by hand.) With no special constraints at all, we can get solutions as follows.

```python
solns= problem.getSolutions()
print(solns)
```

You will notice there are rather a lot of them.

Now formulate the above four special constraints using the Python `constraint` module. To help you, we have done the first one for you.

```
# Olga is leaving 2 hours before the traveller from Yemen.
for person in people:
    problem.addConstraint(
        (lambda x,y,z: (y != "yemen") or
            ((x == "4:30") and (z == "2:30")) or
            ((x == "5:30") and (z == "3:30"))),
        ["t_"+person, "d_"+person, "t_olga"])
```

Note the use of a lambda-expression here to express a constraint. This is a function of three variables, `x`, `y` and `z`, which returns a Boolean value depending on the values of those variables. (You should be able to read the body of the function.) When, in the for-loop, `person` evaluates to — say — `claude`, the `addConstraint` method applies the constraint in question to the triple of variables $\langle$`t_claude`, `d_claude`, `t_olga`$\rangle$, allowing solutions to assign only those tuples of respective values $\langle$`x`, `y`, `z`$\rangle$ to which the function returns `true`.

The other special constraints are handled similarly. Probably the most puzzling is the fourth. As a hint, you might read it as a collection of statements to the effect that certain descriptions do not co-refer: Pablo is not flying from Yemen and is leaving at neither 2:30 nor 3:30; and whoever *is* flying from Yemen is likewise leaving at neither 2:30 nor 3:30. When you have got everything working, you will see that there is in fact just one solution:

```
[{'t_olga': '2:30', 'd_olga': 'peru',
    't_claude': '3:30', 'd_claude': 'romania',
    'd_pablo': 'taiwan', 'd_scott': 'yemen',
    't_scott': '4:30', 't_pablo': '5:30'}]
```

The order of printing is a bit **random**, but do not worry about this.

If you remove the first constraint (the one about Olga leaving two hours before the traveller from Yemen), you will see that there are more solutions...

Task 1: Write a function `Travellers(List)` which takes a list of pairs. Each pair consists of a traveller and a time, e.g. `["olga","2:30"]`, interpreted as the constraint that Olga travels at 2:30. Your function needs to set the logic puzzle with the special constraints 2 to 5 above together with the extra constraints passed as the argument `List`, and return the list of solutions obtained using `constraint.Problem.getSolutions()` that satisfy the resulting system.

For intance, the call to `Travellers([["olga", "2:30"]])` should return the list with one solution given above (possibly with a different order for the problem variables), while the call to `Travellers([["olga", "5:30"]])` should return an empty list as there are no solutions, etc.

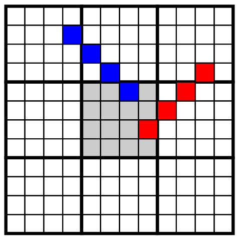
Evaluation: There are 6 marks for this task of the exercise. Your function will be tested on various test cases, and the proportion of fully correct return values will determine your score (out of 6). As explained, for some test cases, there may be no solution (in which case your function should return an empty list).

Pan-diagonal magic squares

Let n be a positive integer. A *magic square* of size n is an $n \times n$ list whose entries are the integers 1 to n^2 arranged in such a way that the sum of each row, each column and both diagonals is the same. Fig. 1(a) shows an example with $n = 4$, in which the entries in every row, every column and the two diagonals add up to 34. Fig. 1(b) shows the same 4×4 square repeated in a 3×3 super-grid. Consider the central 4×4 square (shaded), and starting at any boundary cell, suppose we proceed diagonally (in any of the four directions) for a distance of 4 squares: two such diagonal paths are shown (in blue and red). Since the central square is repeated, these two diagonal paths are, in effect *broken diagonals* of the original square, as shown in Fig. 1(c). Observe that the two main diagonals are just a special case of broken diagonals. A simple check shows that the *broken* diagonals of the square in Fig. 1(a) do not all add up to 34. A *pan-diagonal magic square* of size n is an $n \times n$ list whose entries are the integers 1 to n^2 arranged in such a way that the sum of each row, each column and each broken diagonal is the same. Somewhat surprisingly, pan-diagonal magic squares exist: Fig. 2 shows a pan-diagonal magic square for $n = 4$.

11	2	5	16
10	8	3	13
7	9	14	4
6	15	12	1

(a) A magic square.



(b) Two diagonals.

11	🟥 2	🟦 5	16
🟥 10	8	3	🟦 13
🟦 7	9	14	🟥 4
6	🟦 15	🟥 12	1

(c) Broken diagonals.

Figure 1: Magic squares and broken diagonals.

13	12	7	2
8	1	14	11
10	15	4	5
3	6	9	16

Figure 2: A pan-diagonal magic square of size 4

Task 2: The common sum of each row, column and broken diagonal in a pan-diagonal magic square of size n must be the same. Write a python function `CommonSum(`n`)` to compute this number.

Evaluation: There are 2 marks for this task of the exercise. Your function will be tested automatically on a number of cases, and the proportion of fully correct return values will determine your score (out of 2).

The following Python code sets up the basic framework for finding magic squares of size n.

```
from constraint import Problem,
    AllDifferentConstraint, ExactSumConstraint
problem= Problem()
problem.addVariables(range(0,n*n), range(1,n*n+1))
```

Thus, we have n^2 variables (numbered 0 to $n{-}1$), each of which takes a value in the range 1 to n^2 (inclusive). For definiteness we imagine the variables numbered row by row — i.e. $0, \ldots, n{-}1$ along the bottom row, $n, \ldots, 2n{-}1$ along the next row up, and so on, with $n^2 - n, \ldots, n^2{-}1$ on the top row. We need to say that the variables all take different values. Remember, the constraint package contains a short-cut to do this:

```
problem.addConstraint(AllDifferentConstraint(), range(0,n*n))
```

Next, we need constraints stating that all rows, all columns and the two diagonals have to add up the same number. To help you, here is the constraint for rows.

```
for row in range(n):
    problem.addConstraint(ExactSumConstraint(CommonSum(n)),
        [row * n + i for i in range(n)])
```

Write the corresponding constraint that the columns have to add up to the same number, and also the constraint that the two main diagonals have to add up to this number. Now put everything together so that you compute the total number of magic squares of any given size. You will probably want to use lines like

```
solns= problem.getSolutions()
print(solns)
```

So that we can test your code, we need to add some extra constraints...

Task 3: Write a function `msqList(n, pairList)` which, when `n` is passed an integer n and `pairList` is passed a (possibly empty) list of pairs `[`v`, `i`]` with $0 \leq v < n^2$ and $1 \leq i \leq n^2$, returns the list of *magic squares* of size n such that, for every pair `[`v`, `i`]` in the list, position v is filled with the integer i. Again, you need to use `constraint.Problem.getSolutions()` to obtain this list of solutions.

Remember that the positions in the magic square are assumed to be numbered from 0 to $n^2 - 1$, row by row. For example, `msqList(4,[[0,13],[1,12],[2,7]])` should return a list with 4 solutions, all of which include

```
{0: 13, 1: 12, 2: 7}
```

On the other hand `msqList(3,[])` — i.e. with an empty list of fixed positions — should return a list with 8 solutions. Do not worry if your program is very slow for $n > 4$ and a small (or especially empty) list of extra constraints; we will test on examples which should run in reasonable time.

Evaluation: There are 6 marks for this task of the exercise. Your function will be tested on various test cases, and the proportion of fully correct return values will determine your score (out of 6). Note that, for some test cases, there may be no solution (in which case your function should return an empty list).

Now write two constraints saying that the broken diagonals have to add up to the same number. This is much trickier. As a hint, we recommend that you create a list, say `dd`, such that each member of `dd` is a list of those variables lying along some broken diagonal.

For example, if $n = 4$, `dd` would contain the lists `[0,5,10,15]` and `[4,9,14,3]`.

So that we can test your code, we again need to add extra constraints...

Task 4: Write a function `pmsList(n, pairList)` which, when `n` is passed an integer n and `pairList` is passed a (possibly empty) list of pairs `[`v`, `i`]` with $0 \leq v < n^2$ and $1 \leq i \leq n^2$, returns the list of *pan-diagonal magic squares* of size n such that, for every pair `[`v`, `i`]` in the list, position v is filled with the integer i. You need to use `constraint.Problem.getSolutions()` to obtain this list of solutions.

Again, remember that the positions in the magic square are assumed to be numbered from 0 to $n^2 - 1$, row by row. Thus, one of the solutions that `pmsList(4,[[0,13],[1,12],[2,7]])` should return is the pan-diagonal magic square of Fig. 2, namely

```
{0: 13, 1: 12, 2: 7, 3: 2,
    4: 8, 5: 1, 6: 14, 7: 11,
    8: 10, 9: 15, 10: 4, 11: 5,
    12: 3, 13: 6, 14: 9, 15: 16}
```

Although the actual Python output is unlikely to order positions as was done here. Note incidentally that `pmsList(3,[])` should return `[]` as there are no 3×3 pan-diagonal magic squares.

Also, do not worry if your program is very slow for $n > 4$ and a small (or especially empty) list of extra constraints; we will test on examples which should run in reasonable time.

Evaluation: There are 6 marks for this task of the exercise. Your function will be tested on various test cases, and the proportion of fully correct return values will determine your score (out of 6). Note that, for some test cases, there may be no solution (in which case your function should return an empty list).

[20 marks in total]

Submission

Please follow the README.md instructions in your COMP24011_2022 Gitlab repo to refresh the files of your lab2 branch. You will find a skeleton source file named constraintsLab.py in which to write the Python code for your solution. When you are ready to push your constraintsLab.py remember that you **need to** tag your commit with the command

```
$ git tag 24011-lab2-S-constraints
```

Any other files that you include in your commit **will be ignored**. General instructions on coursework submission using Gitlab can be found on the CS Handbook.

The deadline for submission is **18:00 on Monday, 31st October**. The lab will be **auto-marked** offline. The marking program will download your code from Gitlab and test it against a reference implementation. (This will take time so expect marking to take at least 2 weeks.)

Important: Note that the skeleton constraintsLab.py ends with

```
# Debug
if __name__ == '__main__':
    print("debug run...")
```

You should be familiar with this construction, which allows the source file to be run for debug purposes as well as being imported like a Python module[1]. Apart from the required function definitions **and** any necessary **import** statements, all code that you add in constraintsLab.py **must** be guarded in this way. The reason is that the auto-marking program will test your solution by importing your code as follows

```
from constraintsLab import Travellers, CommonSum, msqList, pmsList
```

and any unguarded leftover code will affect the testing of your functions.

[1]https://docs.python.org/3/library/__main__.html